Akka Stream and HTTP Experimental Scala Documentation

Release 2.0.2

Typesafe Inc

January 14, 2016

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")
```

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

1.2.1 Transforming and consuming simple streams

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

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

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

The `ActorMaterializer` can optionally take `ActorMaterializerSettings` which can be used to define materialization properties, such as default buffer sizes (see also *Buffers 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 – they are "materialized types", which we'll talk about *below*).

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

```
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 `Sink.fold` and `Sink.foreach`):

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

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

The complete snippet looks like this:

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

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

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

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 (Graphs) in order to offer the most convenient API for both of these cases. Graphs can express arbitrarily complex stream setups at the expense of not reading as familiarly as collection transformations.

Graphs are constructed using `GraphDSL` like this:

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

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

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

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

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

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

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 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 `Sink.fold` and see how the types look like:

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

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

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

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

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

First we prepare a reusable `Flow` that will change each incoming tweet into an integer of value `1`. We'll use this in order to combine those ones with a `Sink.fold` will sum all `Int` elements of the stream and make its result available as a `Future[Int]`. 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 `Mat` type parameters on `Source[+Out, +Mat]`, `Flow[-In, +Out, +Mat]` and `Sink[-In, +Mat]`? They represent the type of values these processing parts return when materialized. When you chain these together, you can explicitly combine their materialized values: in our example we used the `Keep.right` predefined function, which tells the implementation to only care about the materialized type of the stage currently appended to the right. As you can notice, the materialized type of sumSink is `Future[Int]` and because of using `Keep.right`, the resulting `RunnableGraph` has also a type parameter of `Future[Int]`.

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

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

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

```
    .toMat(sumSink)(Keep.right)

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

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

Akka Streams does not send dropped stream elements to the dead letter office

One important consequence of offering only features that can be relied upon is the restriction that Akka Streams cannot ensure that all objects sent through a processing topology will be processed. Elements can be dropped for a number of reasons:

- plain user code can consume one element in a *map(...)* stage and produce an entirely different one as its result
- common stream operators drop elements intentionally, e.g. take/drop/filter/conflate/buffer/...

- stream failure will tear down the stream without waiting for processing to finish, all elements that are in flight will be discarded
- stream cancellation will propagate upstream (e.g. from a *take* operator) leading to upstream processing steps being terminated without having processed all of their inputs

This means that sending JVM objects into a stream that need to be cleaned up will require the user to ensure that this happens outside of the Akka Streams facilities (e.g. by cleaning them up after a timeout or when their results are observed on the stream output, or by using other means like finalizers etc.).

Resulting Implementation Constraints

Compositionality entails reusability of partial stream topologies, which led us to the lifted approach of describing data flows as (partial) graphs 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 graphs. 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 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 produces a result termed the *materialized value of a graph*.

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 from the user-level API because we regard them to be an SPI that is not targeted at endusers. In order to obtain a `Publisher` or `Subscriber` from an Akka Stream topology, a corresponding `Sink.asPublisher` or `Source.asSubscriber` 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.asPublisher(true)` (for enabling fan-out support) must be used where broadcast 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, i.e. expose factories that return graphs, allowing full compositionality
- libraries may optionally and additionally provide facilities that consume and materialize graphs

The reasoning behind the first rule is that compositionality would be destroyed if different libraries only accepted graphs 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).

Exceptions from this need to be well-justified and carefully documented.

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
- BidiFlow: 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 stream 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.

1.4 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*.

Non-Blocking Means that a certain operation does not hinder the progress of the calling thread, even if it takes long time to finish the requested operation.

Graph A description of a stream processing topology, defining the pathways through which elements shall flow when the stream is running.

Processing Stage The common name for all building blocks that build up a Graph. 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*

When we talk about *asynchronous, non-blocking backpressure* we mean that the processing stages available in Akka Streams will not use blocking calls but asynchronous message passing to exchange messages between each other, and they will use asynchronous means to slow down a fast producer, without blocking its thread. This is a thread-pool friendly design, since entities that need to wait (a fast producer waiting on a slow consumer) will not block the thread but can hand it back for further use to an underlying thread-pool.

1.4.2 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.

RunnableGraph 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 `RunnableGraph` type, indicating that it is ready to be executed.

It is important to remember that even after constructing the `RunnableGraph` 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 Graph (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 RunnableGraph
val runnable: RunnableGraph[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 `RunnableGraph[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 RunnableGraph
val sink = Sink.fold[Int, Int](0)(_ + _)
val runnable: RunnableGraph[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.fromFuture(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)

// Broadcast to a sink inline
val otherSink: Sink[Int, Unit] =
  Flow[Int].alsoTo(Sink.foreach(println(_))).to(Sink.ignore)
```

```
Source(1 to 6).to(otherSink)
```

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`.

1.4.3 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.

1.4.4 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 `Source` and `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. The actual stream processing is handled by actors 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 `ActorMaterializer`.

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

Operator Fusion

Akka Streams 2.0 contains an initial version of stream operator fusion support. This means that the processing steps of a flow or stream graph can be executed within the same Actor and has three consequences:

- starting up a stream may take longer than before due to executing the fusion algorithm
- passing elements from one processing stage to the next is a lot faster between fused stages due to avoiding the asynchronous messaging overhead
- fused stream processing stages do no longer run in parallel to each other, meaning that only up to one CPU core is used for each fused part

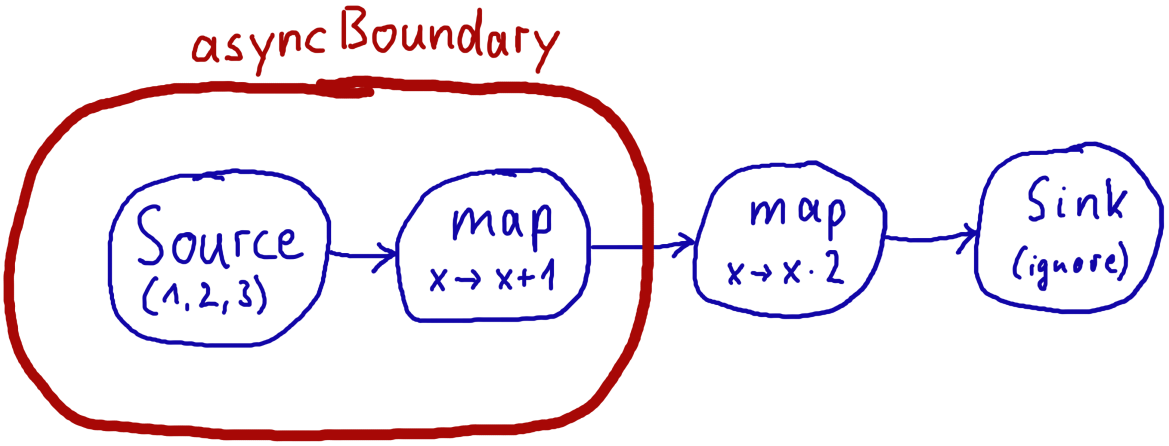
The first point can be countered by pre-fusing and then reusing a stream blueprint as sketched below:

```
import akka.stream.Fusing

val flow = Flow[Int].map(_ * 2).filter(_ > 500)
val fused = Fusing.aggressive(flow)

Source.fromIterator { () => Iterator from 0 }
  .via(fused)
  .take(1000)
```

In order to balance the effects of the second and third bullet points you will have to insert asynchronous boundaries manually into your flows and graphs by way of adding `Attributes.asyncBoundary` to pieces that shall communicate with the rest of the graph in an asynchronous fashion.

```
import akka.stream.Attributes.asyncBoundary

Source(List(1, 2, 3))
  .map(_ + 1)
  .withAttributes(asyncBoundary)
  .map(_ * 2)
  .to(Sink.ignore)
```

In this example we create two regions within the flow which will be executed in one Actor each—assuming that adding and multiplying integers is an extremely costly operation this will lead to a performance gain since two CPUs can work on the tasks in parallel. It is important to note that asynchronous boundaries are not singular places within a flow where elements are passed asynchronously (as in other streaming libraries), but instead attributes always work by adding information to the flow graph that has been constructed up to this point:



This means that everything that is inside the red bubble will be executed by one actor and everything outside of it by another. This scheme can be applied successively, always having one such boundary enclose the previous ones plus all processing stages that have been added since them.

> **Warning:** Without fusing (i.e. up to version 2.0-M2) each stream processing stage had an implicit input buffer that holds a few elements for efficiency reasons. If your flow graphs contain cycles then these buffers may have been crucial in order to avoid deadlocks. With fusing these implicit buffers are no longer there, data elements are passed without buffering between fused stages. In those cases where buffering is needed in order to allow the stream to run at all, you will have to insert explicit buffers with the `.buffer()` combinator—typically a buffer of size 2 is enough to allow a feedback loop to function.

The new fusing behavior can be disabled by setting the configuration parameter `akka.stream.materializer.auto-fusing=off`. In that case you can still manually fuse those graphs which shall run on less Actors. With the exception of the `SslTlsStage` and the `groupBy` operator all built-in processing stages can be fused.

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 source that can be signalled explicitly from the outside
val source: Source[Int, Promise[Option[Int]]] = Source.maybe[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: RunnableGraph[Promise[Option[Int]]] = source.via(flow).to(sink)

// Simple selection of materialized values by using Keep.right
val r2: RunnableGraph[Cancellable] = source.viaMat(flow)(Keep.right).to(sink)
val r3: RunnableGraph[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[Option[Int]] = flow.to(sink).runWith(source)
val r6: (Promise[Option[Int]], Future[Int]) = flow.runWith(source, sink)

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

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

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

val r10: RunnableGraph[(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: RunnableGraph[(Promise[Option[Int]], Cancellable, Future[Int])] =
  r9.mapMaterializedValue {
    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(None)
cancellable.cancel()
future.map(_ + 3)

// The result of r11 can be also achieved by using the Graph API
val r12: RunnableGraph[(Promise[Option[Int]], Cancellable, Future[Int])] =
  RunnableGraph.fromGraph(GraphDSL.create(source, flow, sink)((_, _, _)) { implicit builder =>
```

```
    (src, f, dst) =>
      import GraphDSL.Implicits._
      src ~> f ~> dst
      ClosedShape
  })
```

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*.

1.4.5 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 `GraphStage` – 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 Graphs

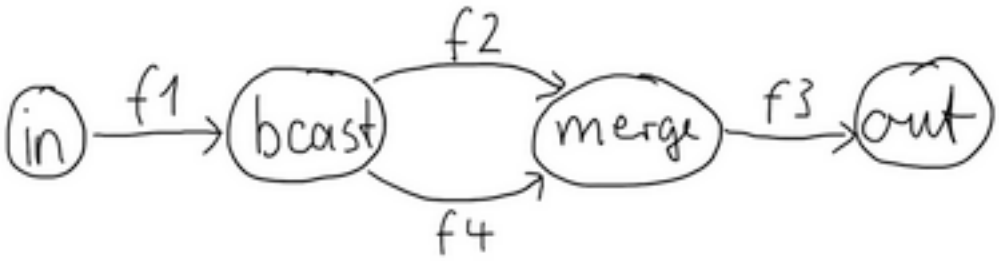
Graphs are built from simple Flows which serve as the linear connections within the graphs as well as junctions which serve as fan-in and fan-out points for Flows. Thanks to the junctions having meaningful types based on their behaviour and making them explicit elements these elements should be rather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

- `UnzipWith[In,A,B,...]` – *(1 input, N outputs)* takes a function of 1 input that given a value for each input emits N output elements (where N <= 20)
- `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`
- **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)

One of the goals of the GraphDSL 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 = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder: GraphDSL.Builder[Unit] =>
  import GraphDSL.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
  ClosedShape
})
```

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

Notice the `import GraphDSL.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).

By looking at the snippets above, it should be apparent that the `GraphDSL.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 GraphDSL has been constructed though, the `GraphDSL` instance *is immutable, thread-safe, and freely shareable*. The same is true of

all graph pieces—sources, sinks, and flows—once they are constructed. This means that you can safely re-use one given Flow or junction 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 `builder.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)

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

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

1.5.2 Constructing and combining Partial 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 by returning a different `Shape` than `ClosedShape`, for example `FlowShape(in, out)`, from the function given to `GraphDSL.create`. See *Predefined shapes*) for a list of such predefined shapes.

Making a `Graph` a `RunnableGraph` 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 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 = GraphDSL.create() { implicit b =>
  import GraphDSL.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 = RunnableGraph.fromGraph(GraphDSL.create(resultSink) { implicit b =>
  sink =>
```

```
    import GraphDSL.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.in
    ClosedShape
})

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 `GraphDSL` 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 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 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 graph with *exactly one* output, that is it returns a `SourceShape`.
- `Sink` is a partial graph with *exactly one* input, that is it returns a `SinkShape`.
- `Flow` is a partial 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 graph the method `Source.fromGraph` is used, to use it we must have a `Graph[SourceShape, T]`. This is constructed using `GraphDSL.create` and returning a `SourceShape` from the function passed in . The single outlet must be provided to the `SourceShape.of` method and 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.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

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

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

  // expose port
```

```
  SourceShape(zip.out)
})

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

Similarly the same can be done for a `Sink[T]`, using `SinkShape.of` in which case the provided value must be an `Inlet[T]`. For defining a `Flow[T]` we need to expose both an inlet and an outlet:

```
val pairUpWithToString =
  Flow.fromGraph(GraphDSL.create() { implicit b =>
    import GraphDSL.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
    FlowShape(broadcast.in, zip.out)
  })

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

1.5.4 Combining Sources and Sinks with simplified API

There is a simplified API you can use to combine sources and sinks with junctions like: `Broadcast[T]`, `Balance[T]`, `Merge[In]` and `Concat[A]` without the need for using the Graph DSL. The combine method takes care of constructing the necessary graph underneath. In following example we combine two sources into one (fan-in):

```
val sourceOne = Source(List(1))
val sourceTwo = Source(List(2))
val merged = Source.combine(sourceOne, sourceTwo)(Merge(_))

val mergedResult: Future[Int] = merged.runWith(Sink.fold(0)(_ + _))
```

The same can be done for a `Sink[T]` but in this case it will be fan-out:

```
val sendRmotely = Sink.actorRef(actorRef, "Done")
val localProcessing = Sink.foreach[Int](_ => /* do something usefull */ ())

val sink = Sink.combine(sendRmotely, localProcessing)(Broadcast[Int](_))

Source(List(0, 1, 2)).runWith(sink)
```

1.5.5 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(
    jobsIn.carbonCopy(),
    priorityJobsIn.carbonCopy(),
    resultsOut.carbonCopy())

  // 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[In, Out](inlets(0).as[In], inlets(1).as[In], outlets(0).as[Out])
  }
}
```

1.5.6 Predefined shapes

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] = {

    GraphDSL.create() { implicit b ⇒
      import GraphDSL.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 " + _)

RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.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)
  ClosedShape
}).run()
```

1.5.7 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 @uncheckedVariance],
                                                     out1: Outlet[Out1 @uncheckedVariance],
                                                     in2: Inlet[In2 @uncheckedVariance],
                                                     out2: Outlet[Out2 @uncheckedVariance]) extend
  // 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.fromGraph(GraphDSL.create() { 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.fromFlows(outbound, inbound)
})

// this is the same as the above
val codec = BidiFlow.fromFunctions(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 `GraphStage` (see also *Custom processing with GraphStage*).

```
val framing = BidiFlow.fromGraph(GraphDSL.create() { 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.fromFlows(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.

1.5.8 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.materializedValue` 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 `materializedValue` any number of times to acquire the necessary number of outlets.

```
import GraphDSL.Implicits._
val foldFlow: Flow[Int, Int, Future[Int]] = Flow.fromGraph(GraphDSL.create(Sink.fold[Int, Int](0)
  implicit builder ⇒
    fold ⇒
      FlowShape(fold.in, builder.materializedValue.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 GraphDSL.Implicits._
// This cannot produce any value:
val cyclicFold: Source[Int, Future[Int]] = Source.fromGraph(GraphDSL.create(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.materializedValue.mapAsync(4)(identity) ~> fold
      SourceShape(builder.materializedValue.mapAsync(4)(identity).outlet)
})
```

1.5.9 Graph cycles, liveness and deadlocks

Cycles in bounded stream topologies 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 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!
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.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
  ClosedShape
})
```

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
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.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
  ClosedShape
})
```

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`.

```
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.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
  ClosedShape
})
```

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
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.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
  ClosedShape
})
```

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`.

```
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.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
  ClosedShape
})
```

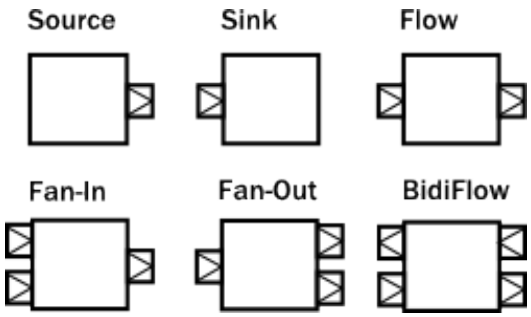
When we run the above example we see that processing starts and never stops. The important takeaway from this example is that balanced cycles often need an initial "kick-off" element to be injected into the cycle.

1.6 Modularity, Composition and Hierarchy

Akka Streams provide a uniform model of stream processing graphs, which allows flexible composition of reusable components. In this chapter we show how these look like from the conceptual and API perspective, demonstrating the modularity aspects of the library.

1.6.1 Basics of composition and modularity

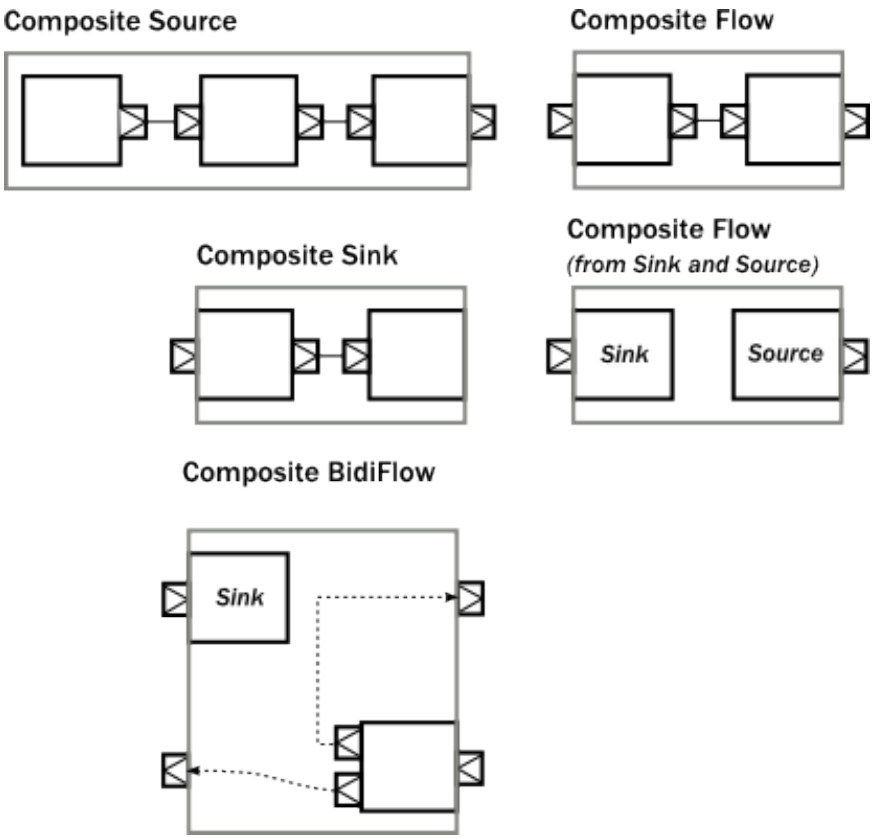
Every processing stage used in Akka Streams can be imagined as a "box" with input and output ports where elements to be processed arrive and leave the stage. In this view, a `Source` is nothing else than a "box" with a single output port, or, a `BidiFlow` is a "box" with exactly two input and two output ports. In the figure below we illustrate the most common used stages viewed as "boxes".



The *linear* stages are `Source`, `Sink` and `Flow`, as these can be used to compose strict chains of processing stages. Fan-in and Fan-out stages have usually multiple input or multiple output ports, therefore they allow to build more complex graph layouts, not just chains. `BidiFlow` stages are usually useful in IO related tasks, where there are input and output channels to be handled. Due to the specific shape of `BidiFlow` it is easy to stack them on top of each other to build a layered protocol for example. The `TLS` support in Akka is for example implemented as a `BidiFlow`.

These reusable components already allow the creation of complex processing networks. What we have seen so far does not implement modularity though. It is desirable for example to package up a larger graph entity into a reusable component which hides its internals only exposing the ports that are meant to the users of the module to interact with. One good example is the `Http` server component, which is encoded internally as a `BidiFlow` which interfaces with the client TCP connection using an input-output port pair accepting and sending `ByteString` s, while its upper ports emit and receive `HttpRequest` and `HttpResponse` instances.

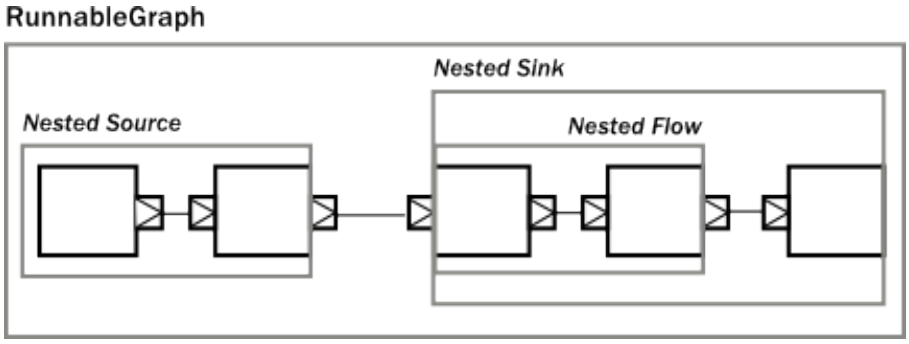
The following figure demonstrates various composite stages, that contain various other type of stages internally, but hiding them behind a *shape* that looks like a `Source`, `Flow`, etc.



One interesting example above is a `Flow` which is composed of a disconnected `Sink` and `Source`. This can be achieved by using the `wrap()` constructor method on `Flow` which takes the two parts as parameters.

The example `BidiFlow` demonstrates that internally a module can be of arbitrary complexity, and the exposed ports can be wired in flexible ways. The only constraint is that all the ports of enclosed modules must be either connected to each other, or exposed as interface ports, and the number of such ports needs to match the requirement of the shape, for example a `Source` allows only one exposed output port, the rest of the internal ports must be properly connected.

These mechanics allow arbitrary nesting of modules. For example the following figure demonstrates a `RunnableGraph` that is built from a composite `Source` and a composite `Sink` (which in turn contains a composite `Flow`).



The above diagram contains one more shape that we have not seen yet, which is called `RunnableGraph`. It turns out, that if we wire all exposed ports together, so that no more open ports remain, we get a module that is *closed*. This is what the `RunnableGraph` class represents. This is the shape that a `Materializer` can take and turn into a network of running entities that perform the task described. In fact, a `RunnableGraph` is a module itself, and (maybe somewhat surprisingly) it can be used as part of larger graphs. It is rarely useful to embed a closed graph shape in a larger graph (since it becomes an isolated island as there are no open port for communication with the rest of the graph), but this demonstrates the uniform underlying model.

If we try to build a code snippet that corresponds to the above diagram, our first try might look like this:

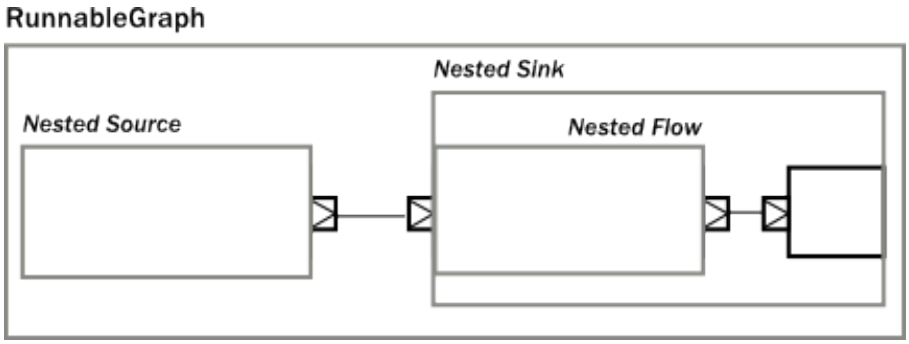
```
Source.single(0)
  .map(_ + 1)
  .filter(_ != 0)
  .map(_ - 2)
  .to(Sink.fold(0)(_ + _))

// ... where is the nesting?
```

It is clear however that there is no nesting present in our first attempt, since the library cannot figure out where we intended to put composite module boundaries, it is our responsibility to do that. If we are using the DSL provided by the `Flow`, `Source`, `Sink` classes then nesting can be achieved by calling one of the methods `withAttributes()` or `named()` (where the latter is just a shorthand for adding a name attribute).

The following code demonstrates how to achieve the desired nesting:

```
val nestedSource =
  Source.single(0) // An atomic source
    .map(_ + 1) // an atomic processing stage
    .named("nestedSource") // wraps up the current Source and gives it a name

val nestedFlow =
  Flow[Int].filter(_ != 0) // an atomic processing stage
    .map(_ - 2) // another atomic processing stage
    .named("nestedFlow") // wraps up the Flow, and gives it a name

val nestedSink =
  nestedFlow.to(Sink.fold(0)(_ + _)) // wire an atomic sink to the nestedFlow
    .named("nestedSink") // wrap it up

// Create a RunnableGraph
val runnableGraph = nestedSource.to(nestedSink)
```

Once we have hidden the internals of our components, they act like any other built-in component of similar shape. If we hide some of the internals of our composites, the result looks just like if any other predefine component has been used:



If we look at usage of built-in components, and our custom components, there is no difference in usage as the code snippet below demonstrates.

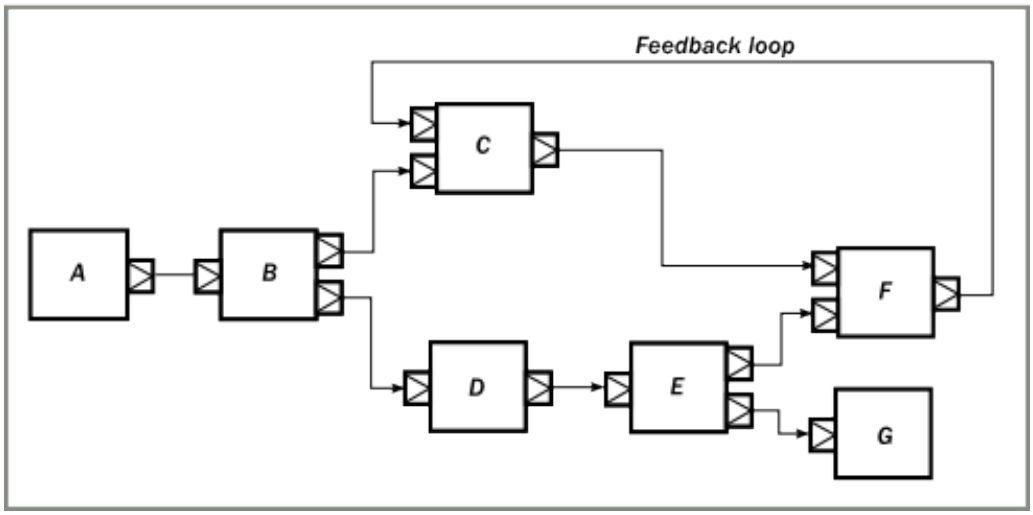
```
// Create a RunnableGraph from our components
val runnableGraph = nestedSource.to(nestedSink)

// Usage is uniform, no matter if modules are composite or atomic
val runnableGraph2 = Source.single(0).to(Sink.fold(0)(_ + _))
```

1.6.2 Composing complex systems

In the previous section we explored the possibility of composition, and hierarchy, but we stayed away from non-linear, generalized graph components. There is nothing in Akka Streams though that enforces that stream processing layouts can only be linear. The DSL for `Source` and friends is optimized for creating such linear chains, as they are the most common in practice. There is a more advanced DSL for building complex graphs, that can be used if more flexibility is needed. We will see that the difference between the two DSLs is only on the surface: the concepts they operate on are uniform across all DSLs and fit together nicely.

As a first example, let's look at a more complex layout:

RunnableGraph



The diagram shows a `RunnableGraph` (remember, if there are no unwired ports, the graph is closed, and therefore can be materialized) that encapsulates a non-trivial stream processing network. It contains fan-in, fan-out stages, directed and non-directed cycles. The `runnable()` method of the `GraphDSL` object allows the creation of a general, closed, and runnable graph. For example the network on the diagram can be realized like this:

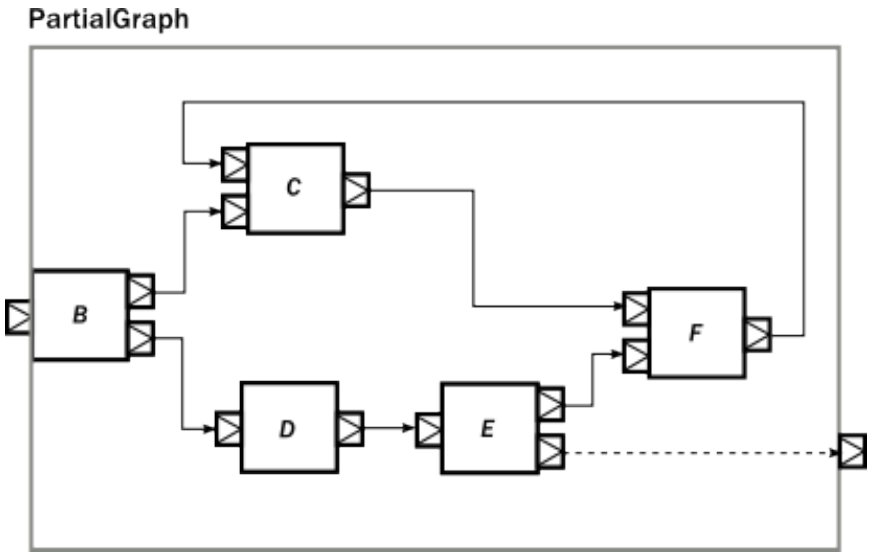
```
import GraphDSL.Implicits._
RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val A: Outlet[Int]                  = builder.add(Source.single(0)).out
  val B: UniformFanOutShape[Int, Int] = builder.add(Broadcast[Int](2))
  val C: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val D: FlowShape[Int, Int]          = builder.add(Flow[Int].map(_ + 1))
  val E: UniformFanOutShape[Int, Int] = builder.add(Balance[Int](2))
  val F: UniformFanInShape[Int, Int]  = builder.add(Merge[Int](2))
  val G: Inlet[Any]                   = builder.add(Sink.foreach(println)).in

                C     <~      F
  A  ~>  B  ~>  C     ~>      F
         B  ~>  D  ~>  E  ~>  F
                       E  ~>  G

  ClosedShape
})
```

In the code above we used the implicit port numbering feature (to make the graph more readable and similar to the diagram) and we imported `Source` s, `Sink` s and `Flow` s explicitly. It is possible to refer to the ports explicitly, and it is not necessary to import our linear stages via `add()`, so another version might look like this:

```
import GraphDSL.Implicits._
RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val B = builder.add(Broadcast[Int](2))
  val C = builder.add(Merge[Int](2))
  val E = builder.add(Balance[Int](2))
  val F = builder.add(Merge[Int](2))

  Source.single(0) ~> B.in; B.out(0) ~> C.in(1); C.out ~> F.in(0)
  C.in(0) <~ F.out

  B.out(1).map(_ + 1) ~> E.in; E.out(0) ~> F.in(1)
```

```
  E.out(1) ~> Sink.foreach(println)
  ClosedShape
})
```

Similar to the case in the first section, so far we have not considered modularity. We created a complex graph, but the layout is flat, not modularized. We will modify our example, and create a reusable component with the graph DSL. The way to do it is to use the `create()` factory method on `GraphDSL`. If we remove the sources and sinks from the previous example, what remains is a partial graph:

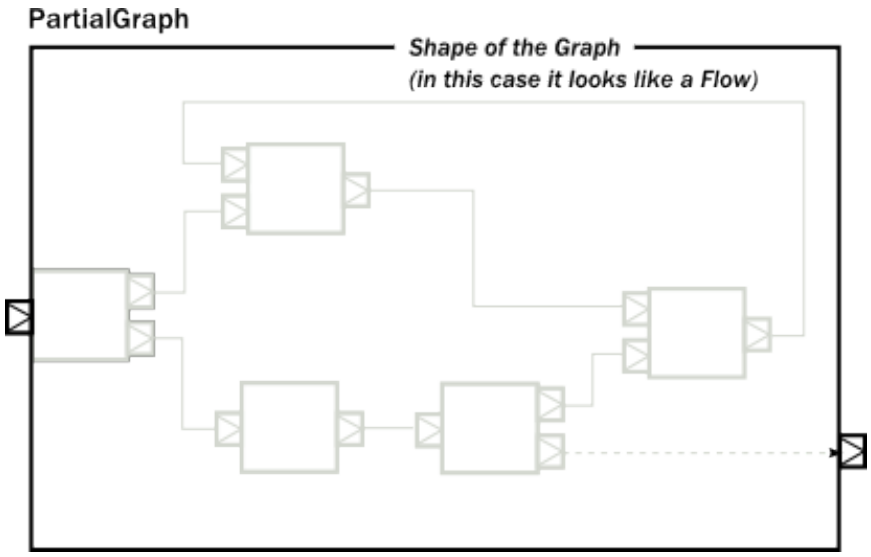


We can recreate a similar graph in code, using the DSL in a similar way than before:

```
import GraphDSL.Implicits._
val partial = GraphDSL.create() { implicit builder =>
  val B = builder.add(Broadcast[Int](2))
  val C = builder.add(Merge[Int](2))
  val E = builder.add(Balance[Int](2))
  val F = builder.add(Merge[Int](2))

                                  C  <~  F
  B  ~>                           C  ~>  F
  B  ~>  Flow[Int].map(_ + 1)  ~>  E  ~>  F
  FlowShape(B.in, E.out(1))
}.named("partial")
```

The only new addition is the return value of the builder block, which is a `Shape`. All graphs (including `Source`, `BidiFlow`, etc) have a shape, which encodes the *typed* ports of the module. In our example there is exactly one input and output port left, so we can declare it to have a `FlowShape` by returning an instance of it. While it is possible to create new `Shape` types, it is usually recommended to use one of the matching built-in ones.

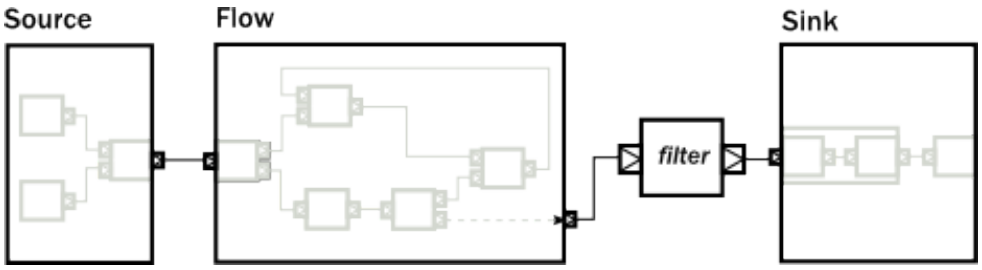
The resulting graph is already a properly wrapped module, so there is no need to call *named()* to encapsulate the graph, but it is a good practice to give names to modules to help debugging.



Since our partial graph has the right shape, it can be already used in the simpler, linear DSL:

```
Source.single(0).via(partial).to(Sink.ignore)
```

It is not possible to use it as a `Flow` yet, though (i.e. we cannot call `.filter()` on it), but `Flow` has a `wrap()` method that just adds the DSL to a `FlowShape`. There are similar methods on `Source`, `Sink` and `BidiShape`, so it is easy to get back to the simpler DSL if a graph has the right shape. For convenience, it is also possible to skip the partial graph creation, and use one of the convenience creator methods. To demonstrate this, we will create the following graph:



The code version of the above closed graph might look like this:

```
// Convert the partial graph of FlowShape to a Flow to get
// access to the fluid DSL (for example to be able to call .filter())
val flow = Flow.fromGraph(partial)

// Simple way to create a graph backed Source
val source = Source.fromGraph( GraphDSL.create() { implicit builder =>
  val merge = builder.add(Merge[Int](2))
  Source.single(0)      ~> merge
  Source(List(2, 3, 4)) ~> merge

  // Exposing exactly one output port
  SourceShape(merge.out)
})

// Building a Sink with a nested Flow, using the fluid DSL
val sink = {
  val nestedFlow = Flow[Int].map(_ * 2).drop(10).named("nestedFlow")
  nestedFlow.to(Sink.head)
}

// Putting all together
val closed = source.via(flow.filter(_ > 1)).to(sink)
```

Note: All graph builder sections check if the resulting graph has all ports connected except the exposed ones and will throw an exception if this is violated.

We are still in debt of demonstrating that `RunnableGraph` is a component just like any other, which can be embedded in graphs. In the following snippet we embed one closed graph in another:

```
val closed1 = Source.single(0).to(Sink.foreach(println))
val closed2 = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  val embeddedClosed: ClosedShape = builder.add(closed1)
  // ...
  embeddedClosed
})
```

The type of the imported module indicates that the imported module has a `ClosedShape`, and so we are not able to wire it to anything else inside the enclosing closed graph. Nevertheless, this "island" is embedded properly, and will be materialized just like any other module that is part of the graph.

As we have demonstrated, the two DSLs are fully interoperable, as they encode a similar nested structure of "boxes with ports", it is only the DSLs that differ to be as much powerful as possible on the given abstraction level. It is possible to embed complex graphs in the fluid DSL, and it is just as easy to import and embed a `Flow`, etc, in a larger, complex structure.

We have also seen, that every module has a `Shape` (for example a `Sink` has a `SinkShape`) independently which DSL was used to create it. This uniform representation enables the rich composability of various stream processing entities in a convenient way.

1.6.3 Materialized values

After realizing that `RunnableGraph` is nothing more than a module with no unused ports (it is an island), it becomes clear that after materialization the only way to communicate with the running stream processing logic is via some side-channel. This side channel is represented as a *materialized value*. The situation is similar to `Actor`s, where the `Props` instance describes the actor logic, but it is the call to `actorOf()` that creates an actually running actor, and returns an `ActorRef` that can be used to communicate with the running actor itself. Since the `Props` can be reused, each call will return a different reference.

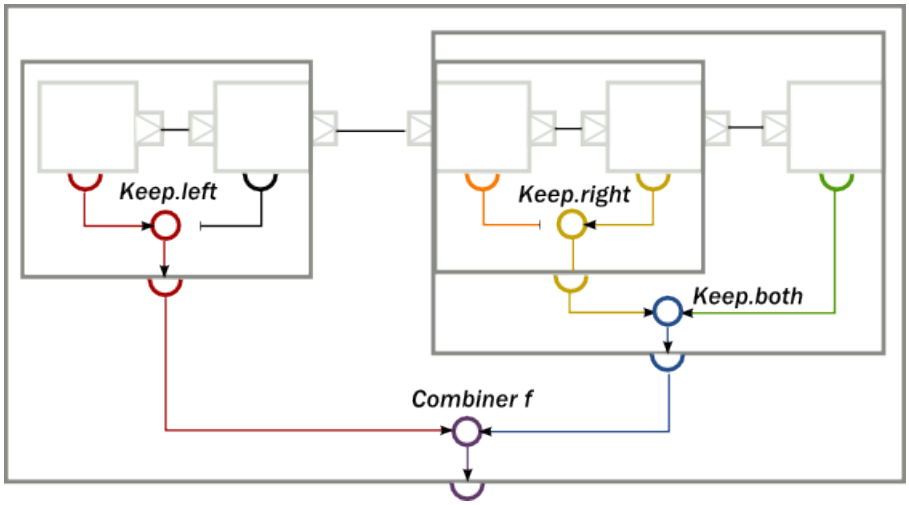
When it comes to streams, each materialization creates a new running network corresponding to the blueprint that was encoded in the provided `RunnableGraph`. To be able to interact with the running network, each

materialization needs to return a different object that provides the necessary interaction capabilities. In other words, the `RunnableGraph` can be seen as a factory, which creates:

- a network of running processing entities, inaccessible from the outside
- a materialized value, optionally providing a controlled interaction capability with the network

Unlike actors though, each of the processing stages might provide a materialized value, so when we compose multiple stages or modules, we need to combine the materialized value as well (there are default rules which make this easier, for example *to()* and *via()* takes care of the most common case of taking the materialized value to the left. See *Combining materialized values* for details). We demonstrate how this works by a code example and a diagram which graphically demonstrates what is happening.

The propagation of the individual materialized values from the enclosed modules towards the top will look like this:



To implement the above, first, we create a composite `Source`, where the enclosed `Source` have a materialized type of `Promise[Unit]`. By using the combiner function `Keep.left`, the resulting materialized type is of the nested module (indicated by the color *red* on the diagram):

```
// Materializes to Promise[Option[Int]]                                 (red)
val source: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]

// Materializes to Unit                                                 (black)
val flow1: Flow[Int, Int, Unit] = Flow[Int].take(100)

// Materializes to Promise[Int]                                         (red)
val nestedSource: Source[Int, Promise[Option[Int]]] =
  source.viaMat(flow1)(Keep.left).named("nestedSource")
```

Next, we create a composite `Flow` from two smaller components. Here, the second enclosed `Flow` has a materialized type of `Future[OutgoingConnection]`, and we propagate this to the parent by using `Keep.right` as the combiner function (indicated by the color *yellow* on the diagram):

```
// Materializes to Unit                                                 (orange)
val flow2: Flow[Int, ByteString, Unit] = Flow[Int].map { i => ByteString(i.toString) }
```

```
// Materializes to Future[OutgoingConnection]                    (yellow)
val flow3: Flow[ByteString, ByteString, Future[OutgoingConnection]] =
  Tcp().outgoingConnection("localhost", 8080)

// Materializes to Future[OutgoingConnection]                    (yellow)
val nestedFlow: Flow[Int, ByteString, Future[OutgoingConnection]] =
  flow2.viaMat(flow3)(Keep.right).named("nestedFlow")
```

As a third step, we create a composite `Sink`, using our `nestedFlow` as a building block. In this snippet, both the enclosed `Flow` and the folding `Sink` has a materialized value that is interesting for us, so we use `Keep.both` to get a `Pair` of them as the materialized type of `nestedSink` (indicated by the color *blue* on the diagram)

```
// Materializes to Future[String]                                (green)
val sink: Sink[ByteString, Future[String]] = Sink.fold("")(_ + _.utf8String)

// Materializes to (Future[OutgoingConnection], Future[String])  (blue)
val nestedSink: Sink[Int, (Future[OutgoingConnection], Future[String])] =
  nestedFlow.toMat(sink)(Keep.both)
```

As the last example, we wire together `nestedSource` and `nestedSink` and we use a custom combiner function to create a yet another materialized type of the resulting `RunnableGraph`. This combiner function just ignores the `Future[Sink]` part, and wraps the other two values in a custom case class `MyClass` (indicated by color *purple* on the diagram):

```
case class MyClass(private val p: Promise[Option[Int]], conn: OutgoingConnection) {
  def close() = p.trySuccess(None)
}

def f(p: Promise[Option[Int]],
      rest: (Future[OutgoingConnection], Future[String])): Future[MyClass] = {

  val connFuture = rest._1
  connFuture.map(MyClass(p, _))
}

// Materializes to Future[MyClass]                               (purple)
val runnableGraph: RunnableGraph[Future[MyClass]] =
  nestedSource.toMat(nestedSink)(f)
```

Note: The nested structure in the above example is not necessary for combining the materialized values, it just demonstrates how the two features work together. See *Combining materialized values* for further examples of combining materialized values without nesting and hierarchy involved.

1.6.4 Attributes

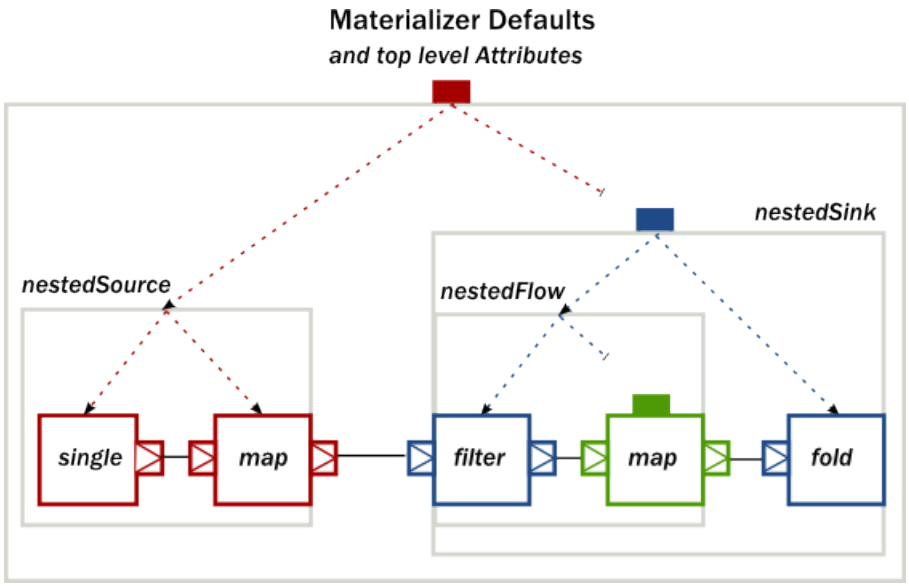
We have seen that we can use `named()` to introduce a nesting level in the fluid DSL (and also explicit nesting by using `create()` from `GraphDSL`). Apart from having the effect of adding a nesting level, `named()` is actually a shorthand for calling `withAttributes(Attributes.name("someName"))`. Attributes provide a way to fine-tune certain aspects of the materialized running entity. For example buffer sizes can be controlled via attributes (see *Buffers in Akka Streams*). When it comes to hierarchic composition, attributes are inherited by nested modules, unless they override them with a custom value.

The code below, a modification of an earlier example sets the `inputBuffer` attribute on certain modules, but not on others:

```
import Attributes._
val nestedSource =
  Source.single(0)
    .map(_ + 1)
    .named("nestedSource") // Wrap, no inputBuffer set
```

```
val nestedFlow =
  Flow[Int].filter(_ != 0)
    .via(Flow[Int].map(_ - 2).withAttributes(inputBuffer(4, 4))) // override
    .named("nestedFlow") // Wrap, no inputBuffer set

val nestedSink =
  nestedFlow.to(Sink.fold(0)(_ + _)) // wire an atomic sink to the nestedFlow
    .withAttributes(name("nestedSink") and inputBuffer(3, 3)) // override
```

The effect is, that each module inherits the `inputBuffer` attribute from its enclosing parent, unless it has the same attribute explicitly set. `nestedSource` gets the default attributes from the materializer itself. `nestedSink` on the other hand has this attribute set, so it will be used by all nested modules. `nestedFlow` will inherit from `nestedSink` except the `map` stage which has again an explicitly provided attribute overriding the inherited one.



This diagram illustrates the inheritance process for the example code (representing the materializer default attributes as the color *red*, the attributes set on `nestedSink` as *blue* and the attributes set on `nestedFlow` as *green*).

1.7 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 is 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 have 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.

1.7.1 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 suitable for the throughput requirements of the application. Default buffer sizes can be set through configuration:

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

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

```
val materializer = ActorMaterializer(
  ActorMaterializerSettings(system)
    .withInputBuffer(
      initialSize = 64,
      maxSize = 64))
```

If the 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(Attributes.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()

RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

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

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

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

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

Running the above example one would expect the number *3* to be printed in every 3 seconds (the `cUndefinedSourceonflate` 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)
```

Instead of dropping the youngest element from the tail of the buffer a new element can be dropped without enqueueing it to the buffer at all.

```
jobs.buffer(1000, OverflowStrategy.dropNew)
```

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)
```

1.7.2 Rate transformation

Understanding conflate

When a fast producer can not be informed to slow down by backpressure or some other signal, `conflate` might be useful to combine elements from a producer until a demand signal comes from a consumer.

Below is an example snippet that summarizes fast stream of elements to a standart deviation, mean and count of elements that have arrived while the stats have been calculated.

```
val statsFlow = Flow[Double]
  .conflate(Seq(_))(_ :+ _)
  .map { s =>
    val μ = s.sum / s.size
    val se = s.map(x => pow(x - μ, 2))
    val σ = sqrt(se.sum / se.size)
    (σ, μ, s.size)
  }
```

This example demonstrates that such flow's rate is decoupled. The element rate at the start of the flow can be much higher that the element rate at the end of the flow.

Another possible use of `conflate` is to not consider all elements for summary when producer starts getting too fast. Example below demonstrates how `conflate` can be used to implement random drop of elements when consumer is not able to keep up with the producer.

```
val p = 0.01
val sampleFlow = Flow[Double]
  .conflate(Seq(_)) {
    case (acc, elem) if Random.nextDouble < p => acc :+ elem
    case (acc, _)                             => acc
  }
  .mapConcat(identity)
```

Understanding expand

Expand helps to deal with slow producers which are unable to keep up with the demand coming from consumers. Expand allows to extrapolate a value to be sent as an element to a consumer.

As a simple use of `expand` here is a flow that sends the same element to consumer when producer does not send any new elements.

```
val lastFlow = Flow[Double]
  .expand(identity)(s => (s, s))
```

Expand also allows to keep some state between demand requests from the downstream. Leveraging this, here is a flow that tracks and reports a drift between fast consumer and slow producer.

```
val driftFlow = Flow[Double]
  .expand((_, 0)) {
    case (lastElement, drift) => ((lastElement, drift), (lastElement, drift + 1))
  }
```

Note that all of the elements coming from upstream will go through `expand` at least once. This means that the output of this flow is going to report a drift of zero if producer is fast enough, or a larger drift otherwise.

1.8 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.

Note: A custom graph stage should not be the first tool you reach for, defining graphs using flows and the graph DSL is in general easier and does to a larger extent protect you from mistakes that might be easy to make with a custom `GraphStage`

1.8.1 Custom processing with GraphStage

The `GraphStage` abstraction can be used to create arbitrary graph processing stages with any number of input or output ports. It is a counterpart of the `GraphDSL.create()` method which creates new stream processing stages by composing others. Where `GraphStage` differs is that it creates a stage that is itself not divisible into smaller ones, and allows state to be maintained inside it in a safe way.

As a first motivating example, we will build a new `Source` that will simply emit numbers from 1 until it is cancelled. To start, we need to define the "interface" of our stage, which is called *shape* in Akka Streams terminology (this is explained in more detail in the section *Modularity, Composition and Hierarchy*). This is how this looks like:

```
import akka.stream.SourceShape
import akka.stream.stage.GraphStage

class NumbersSource extends GraphStage[SourceShape[Int]] {
  // Define the (sole) output port of this stage
  val out: Outlet[Int] = Outlet("NumbersSource")
  // Define the shape of this stage, which is SourceShape with the port we defined above
  override val shape: SourceShape[Int] = SourceShape(out)

  // This is where the actual (possibly stateful) logic will live
  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = ???
}
```

As you see, in itself the `GraphStage` only defines the ports of this stage and a shape that contains the ports. It also has, a currently unimplemented method called `createLogic`. If you recall, stages are reusable in multiple materializations, each resulting in a different executing entity. In the case of `GraphStage` the actual running logic is modeled as an instance of a `GraphStageLogic` which will be created by the materializer by calling the `createLogic` method. In other words, all we need to do is to create a suitable logic that will emit the numbers we want.

In order to emit from a `Source` in a backpressured stream one needs first to have demand from downstream. To receive the necessary events one needs to register a subclass of `OutHandler` with the output port (`Outlet`). This handler will receive events related to the lifecycle of the port. In our case we need to override `onPull()` which indicates that we are free to emit a single element. There is another callback, `onDownstreamFinish()` which is called if the downstream cancelled. Since the default behavior of that callback is to stop the stage, we don't need to override it. In the `onPull` callback we will simply emit the next number. This is how it looks like in the end:

```
import akka.stream.SourceShape
import akka.stream.Graph
import akka.stream.stage.GraphStage
import akka.stream.stage.OutHandler

class NumbersSource extends GraphStage[SourceShape[Int]] {
  val out: Outlet[Int] = Outlet("NumbersSource")
  override val shape: SourceShape[Int] = SourceShape(out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      // All state MUST be inside the GraphStageLogic,
      // never inside the enclosing GraphStage.
      // This state is safe to access and modify from all the
      // callbacks that are provided by GraphStageLogic and the
      // registered handlers.
      private var counter = 1

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          push(out, counter)
          counter += 1
        }
      })
    }
}
```

Instances of the above `GraphStage` are subclasses of `Graph[SourceShape[Int],Unit]` which means that they are already usable in many situations, but do not provide the DSL methods we usually have for other `Source` s. In order to convert this `Graph` to a proper `Source` we need to wrap it using `Source.fromGraph` (see *Modularity, Composition and Hierarchy* for more details about graphs and DSLs). Now we can use the source as any other built-in one:

```
// A GraphStage is a proper Graph, just like what GraphDSL.create would return
val sourceGraph: Graph[SourceShape[Int], Unit] = new NumbersSource

// Create a Source from the Graph to access the DSL
val mySource: Source[Int, Unit] = Source.fromGraph(new NumbersSource)

// Returns 55
val result1: Future[Int] = mySource.take(10).runFold(0)(_ + _)

// The source is reusable. This returns 5050
val result2: Future[Int] = mySource.take(100).runFold(0)(_ + _)
```

Port states, InHandler and OutHandler

In order to interact with a port (`Inlet` or `Outlet`) of the stage we need to be able to receive events and generate new events belonging to the port. From the `GraphStageLogic` the following operations are available on an output port:

- `push(out,elem)` pushes an element to the output port. Only possible after the port has been pulled by downstream.
- `complete(out)` closes the output port normally.
- `fail(out,exception)` closes the port with a failure signal.

The events corresponding to an *output* port can be received in an `OutHandler` instance registered to the output port using `setHandler(out,handler)`. This handler has two callbacks:

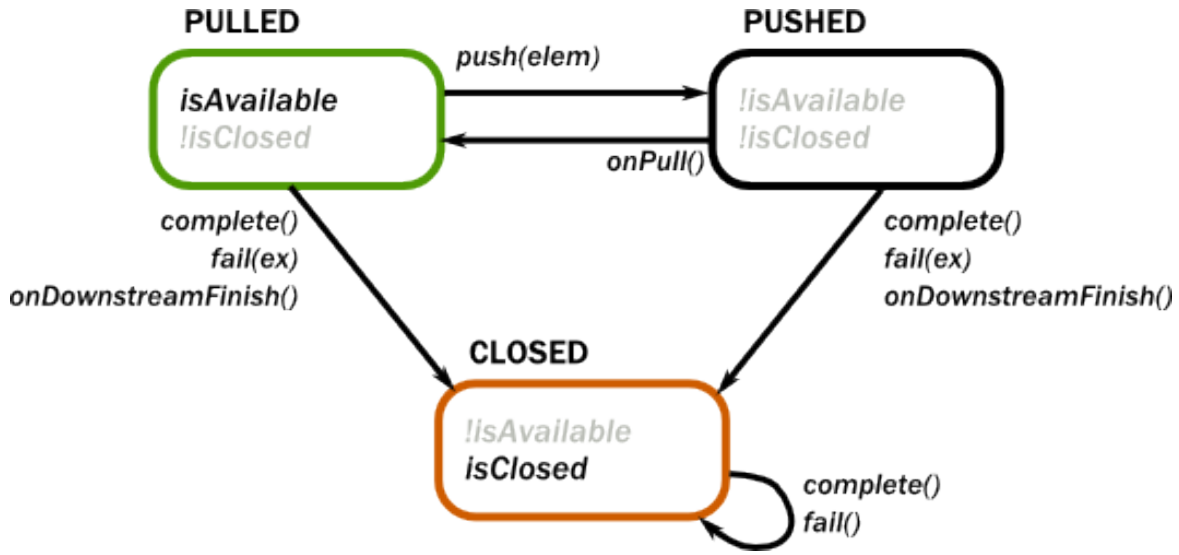
- `onPull()` is called when the output port is ready to emit the next element, `push(out, elem)` is now allowed to be called on this port.

- `onDownstreamFinish()` is called once the downstream has cancelled and no longer allows messages to be pushed to it. No more `onPull()` will arrive after this event. If not overridden this will default to stopping the stage.

Also, there are two query methods available for output ports:

- `isAvailable(out)` returns true if the port can be pushed
- `isClosed(out)` returns true if the port is closed. At this point the port can not be pushed and will not be pulled anymore.

The relationship of the above operations, events and queries are summarized in the state machine below. Green shows the initial state while orange indicates the end state. If an operation is not listed for a state, then it is invalid to call it while the port is in that state. If an event is not listed for a state, then that event cannot happen in that state.



The following operations are available for *input* ports:

- `pull(in)` requests a new element from an input port. This is only possible after the port has been pushed by upstream.
- `grab(in)` acquires the element that has been received during an `onPush()`. It cannot be called again until the port is pushed again by the upstream.
- `cancel(in)` closes the input port.

The events corresponding to an *input* port can be received in an `InHandler` instance registered to the input port using `setHandler(in, handler)`. This handler has three callbacks:

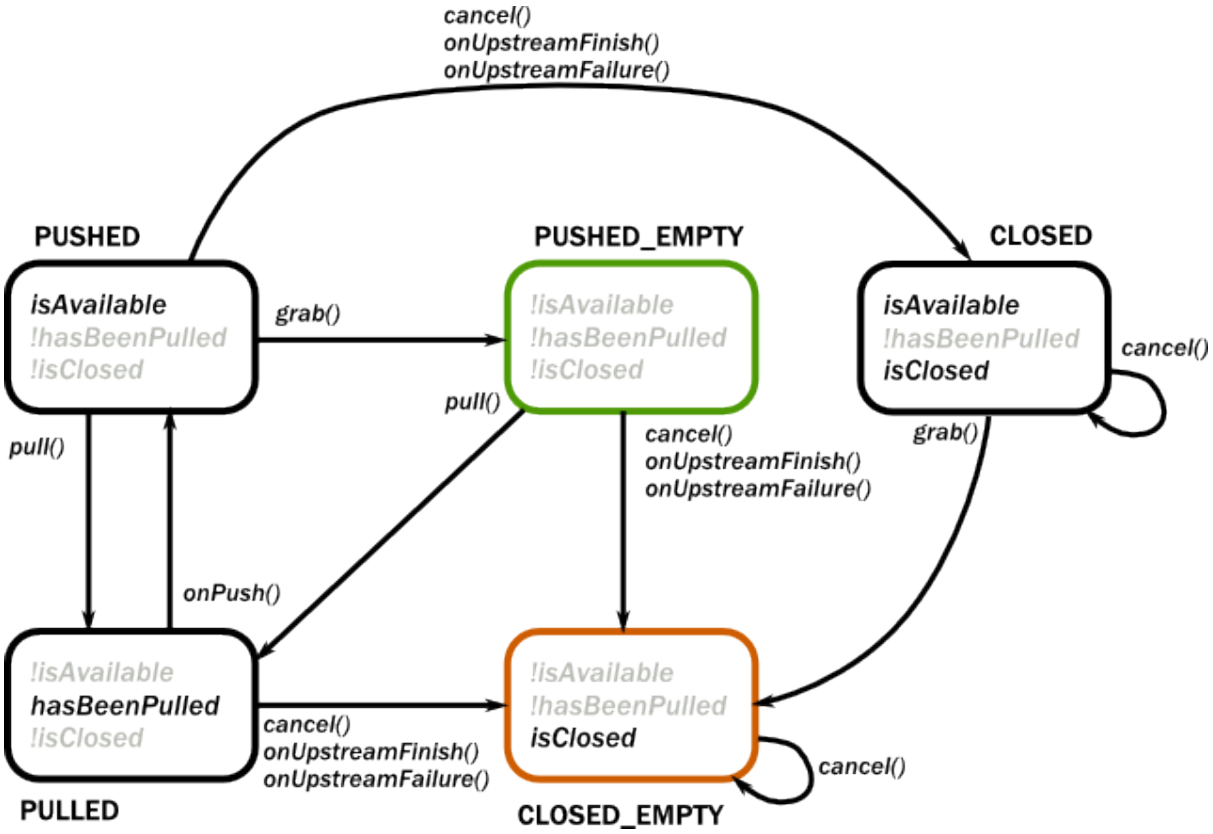
- `onPush()` is called when the output port has now a new element. Now it is possible to aquire this element using `grab(in)` and/or call `pull(in)` on the port to request the next element. It is not mandatory to grab the element, but if it is pulled while the element has not been grabbed it will drop the buffered element.
- `onUpstreamFinish()` is called once the upstream has completed and no longer can be pulled for new elements. No more `onPush()` will arrive after this event. If not overridden this will default to stopping the stage.

- `onUpstreamFailure()` is called if the upstream failed with an exception and no longer can be pulled for new elements. No more `onPush()` will arrive after this event. If not overridden this will default to failing the stage.

Also, there are three query methods available for input ports:

- `isAvailable(in)` returns true if the port can be grabbed.
- `hasBeenPulled(in)` returns true if the port has been already pulled. Calling `pull(in)` in this state is illegal.
- `isClosed(in)` returns true if the port is closed. At this point the port can not be pulled and will not be pushed anymore.

The relationship of the above operations, events and queries are summarized in the state machine below. Green shows the initial state while orange indicates the end state. If an operation is not listed for a state, then it is invalid to call it while the port is in that state. If an event is not listed for a state, then that event cannot happen in that state.



Finally, there are two methods available for convenience to complete the stage and all of its ports:

- `completeStage()` is equivalent to closing all output ports and cancelling all input ports.
- `failStage(exception)` is equivalent to failing all output ports and cancelling all input ports.

In some cases it is inconvenient and error prone to react on the regular state machine events with the signal based API described above. For those cases there is a API which allows for a more declarative sequencing of actions

which will greatly simplify some use cases at the cost of some extra allocations. The difference between the two APIs could be described as that the first one is signal driven from the outside, while this API is more active and drives its surroundings.

The operations of this part of the :class:`GraphStage` API are:

- `emit(out, elem)` and `emitMultiple(out, Iterable(elem1, elem2))` replaces the `OutHandler` with a handler that emits one or more elements when there is demand, and then reinstalls the current handlers
- `read(in)(andThen)` and `readN(in, n)(andThen)` replaces the `InHandler` with a handler that reads one or more elements as they are pushed and allows the handler to react once the requested number of elements has been read.
- `abortEmitting()` and `abortReading()` which will cancel an ongoing emit or read

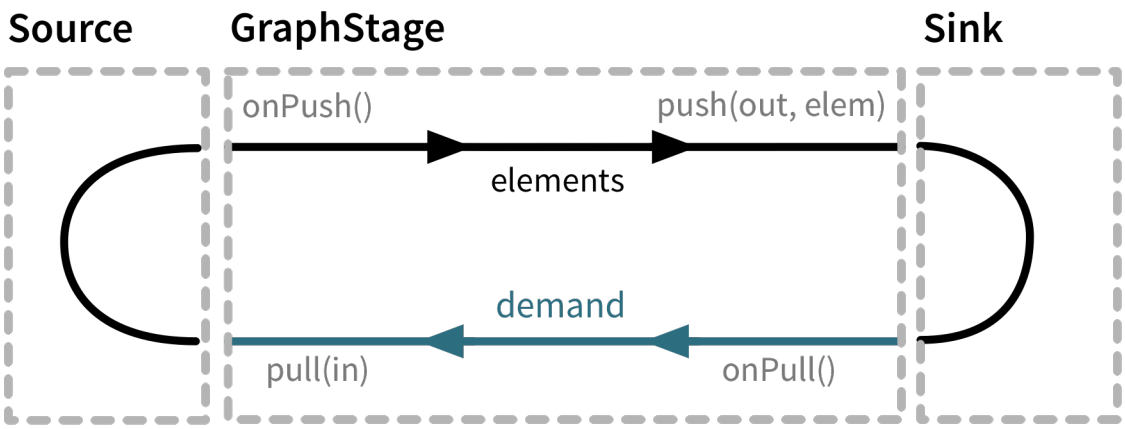
Note that since the above methods are implemented by temporarily replacing the handlers of the stage you should never call `setHandler` while they are running `emit` or `read` as that interferes with how they are implemented. The following methods are safe to call after invoking `emit` and `read` (and will lead to actually running the operation when those are done): `complete(out)`, `completeStage()`, `emit`, `emitMultiple`, `abortEmitting()` and `abortReading()`

An example of how this API simplifies a stage can be found below in the second version of the :class:`Duplicator`.

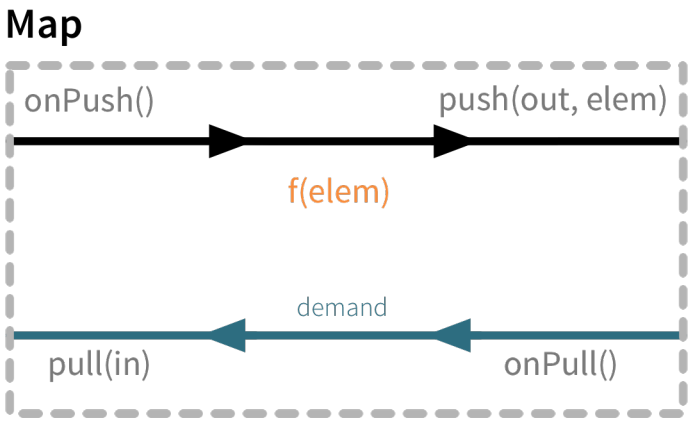
Custom linear processing stages using GraphStage

Graph stages allows for custom linear processing stages through letting them have one input and one output and using `FlowShape` as their shape.

Such a stage can be illustrated as a box with two flows as it is seen in the illustration below. Demand flowing upstream leading to elements flowing downstream.



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

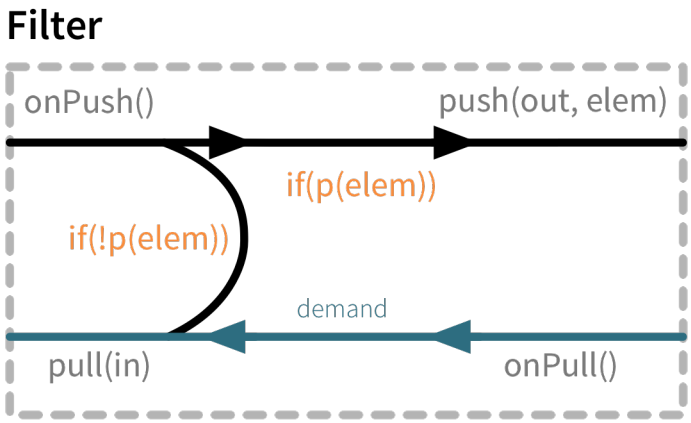


Map calls `push(out)` from the `onPush()` handler and it also calls `pull()` from the `onPull` handler resulting in the conceptual wiring above, and fully expressed in code below:

```
class Map[A, B](f: A => B) extends GraphStage[FlowShape[A, B]] {

  val in = Inlet[A]("Map.in")
  val out = Outlet[B]("Map.out")

  override val shape = FlowShape.of(in, out)

  override def createLogic(attr: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          push(out, f(grab(in)))
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }
}
```

Map is a typical example of a one-to-one transformation of a stream where demand is passed along upstream elements passed on downstream.

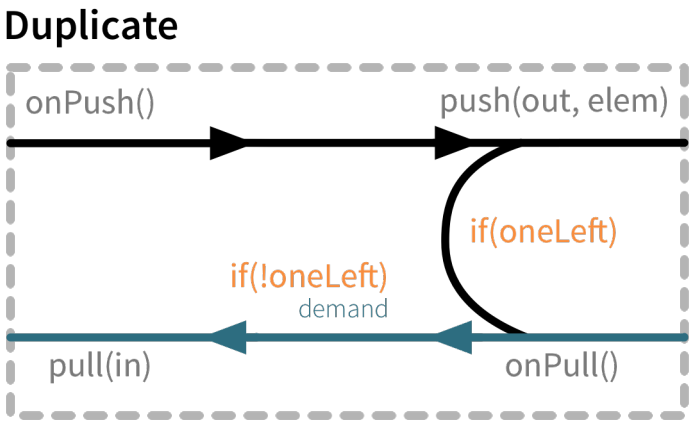
To demonstrate a many-to-one stage we will implement filter. The conceptual wiring of `Filter` looks like this:



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 `pull(in)` or `push(out)` call (and of course not having a mapping `f` function).

```
class Filter[A](p: A => Boolean) extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("Filter.in")
  val out = Outlet[A]("Filter.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          if (p(elem)) push(out, elem)
          else pull(in)
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }
}
```

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: an option with the last element it has seen indicating if it has duplicated this last element already or not. We must also make sure to emit the extra element if the upstream completes.

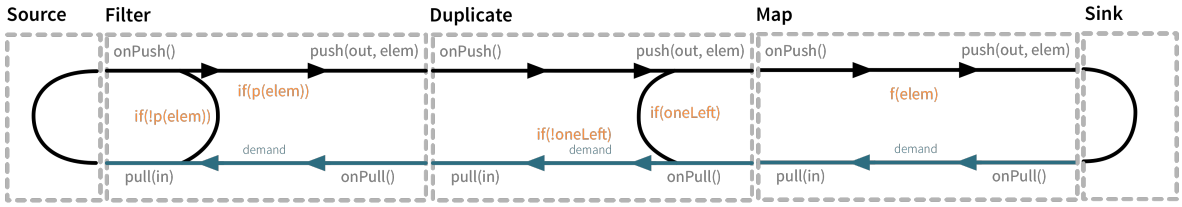
```
class Duplicator[A] extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("Duplicator.in")
  val out = Outlet[A]("Duplicator.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      // Again: note that all mutable state
      // MUST be inside the GraphStageLogic
      var lastElem: Option[A] = None

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          lastElem = Some(elem)
          push(out, elem)
        }

        override def onUpstreamFinish(): Unit = {
          if (lastElem.isDefined) emit(out, lastElem.get)
          complete(out)
        }

      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          if (lastElem.isDefined) {
            push(out, lastElem.get)
            lastElem = None
          } else {
            pull(in)
          }
        }
      })
    }
}
```

In this case a pull from downstream might be consumed by the stage itself rather than passed along upstream as the stage might contain an element it wants to push. Note that we also need to handle the case where the upstream closes while the stage still has elements it wants to push downstream. This is done by overriding *onUpstreamFinish* in the *InHandler* and provide custom logic that should happen when the upstream has been finished.

This example can be simplified by replacing the usage of a mutable state with calls to `emitMultiple` which will replace the handlers, emit each of multiple elements and then reinstate the original handlers:

```
class Duplicator[A] extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("Duplicator.in")
  val out = Outlet[A]("Duplicator.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          // this will temporarily suspend this handler until the two elems
          // are emitted and then reinstates it
          emitMultiple(out, Iterable(elem, elem))
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })
    }
}
```

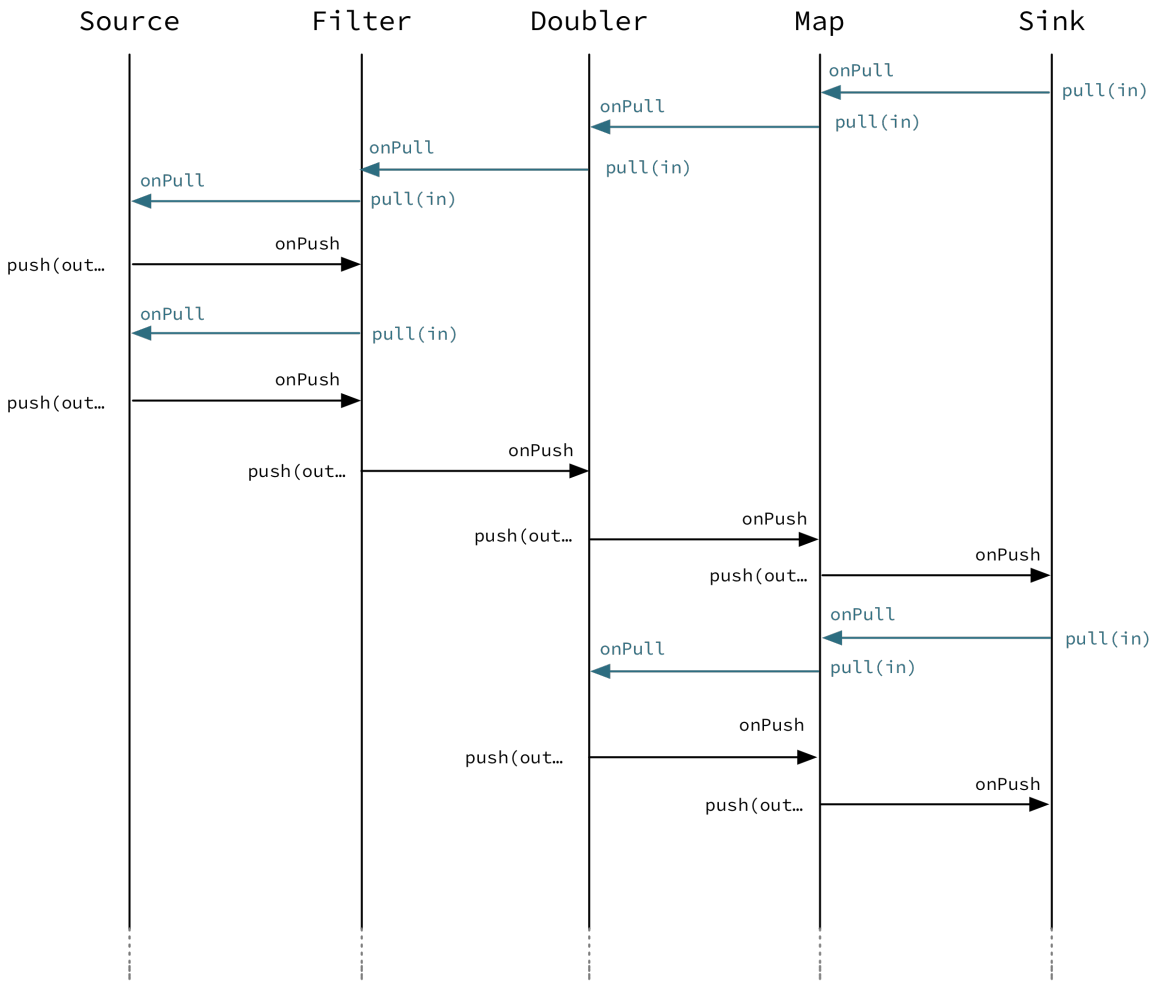
Finally, to demonstrate all of the stages above, we put them together into a processing chain, which conceptually would correspond to the following structure:



In code this is only a few lines, using the `via` use our custom stages in a stream:

```
val resultFuture = Source(1 to 5)
  .via(new Filter(_ % 2 == 0))
  .via(new Duplicator())
  .via(new Map(_ / 2))
  .runWith(sink)
```

If we attempt to draw the sequence of events, it shows that there is one "event token" in circulation in a potential chain of stages, just like our conceptual "railroad tracks" representation predicts.



Completion

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 first `Duplicator` implementation where the last element needs to be doubled even after the upstream neighbor stage has been completed. This can be done by overriding the `onUpstreamFinish` method in `InHandler`.

Stages by default automatically stop once all of their ports (input and output) have been closed externally or internally. It is possible to opt out from this behavior by invoking `setKeepGoing(true)` (which is not supported from the stage's constructor and usually done in `preStart`). In this case the stage **must** be explicitly closed by calling `completeStage()` or `failStage(exception)`. This feature carries the risk of leaking streams and actors, therefore it should be used with care.

Using timers

It is possible to use timers in `GraphStages` by using `TimerGraphStageLogic` as the base class for the returned logic. Timers can be scheduled by calling one of `scheduleOnce(key,delay)`, `schedulePeriodically(key,period)` or `schedulePeriodicallyWithInitialDelay(key,delay,period)` and passing an object as a key for that timer (can be any object, for example a `String`). The `onTimer(key)` method needs to be overridden and it will be called once the timer of `key` fires. It is possible to cancel a timer using `cancelTimer(key)` and check the status of a timer with `isTimerActive(key)`. Timers will be automatically cleaned up when the stage completes.

Timers can not be scheduled from the constructor of the logic, but it is possible to schedule them from the `preStart()` lifecycle hook.

In this sample the stage toggles between open and closed, where open means no elements are passed through. The stage starts out as closed but as soon as an element is pushed downstream the gate becomes open for a duration of time during which it will consume and drop upstream messages:

```
// each time an event is pushed through it will trigger a period of silence
class TimedGate[A](silencePeriod: FiniteDuration) extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("TimedGate.in")
  val out = Outlet[A]("TimedGate.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new TimerGraphStageLogic(shape) {

      var open = false

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          if (open) pull(in)
          else {
            push(out, elem)
            open = true
            scheduleOnce(None, silencePeriod)
          }
        }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = { pull(in) }
      })

      override protected def onTimer(timerKey: Any): Unit = {
        open = false
      }
    }
}
```

Using asynchronous side-channels

In order to receive asynchronous events that are not arriving as stream elements (for example a completion of a future or a callback from a 3rd party API) one must acquire a `AsyncCallback` by calling `getAsyncCallback()` from the stage logic. The method `getAsyncCallback` takes as a parameter a callback that will be called once the asynchronous event fires. It is important to **not call the callback directly**, instead, the external API must call the `invoke(event)` method on the returned `AsyncCallback`. The execution engine will take care of calling the provided callback in a thread-safe way. The callback can safely access the state of the `GraphStageLogic` implementation.

Sharing the AsyncCallback from the constructor risks race conditions, therefore it is recommended to use the `preStart()` lifecycle hook instead.

This example shows an asynchronous side channel graph stage that starts dropping elements when a future completes:

```
// will close upstream when the future completes
class KillSwitch[A](switch: Future[Unit]) extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("KillSwitch.in")
  val out = Outlet[A]("KillSwitch.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {

      override def preStart(): Unit = {
        val callback = getAsyncCallback[Unit] { (_) =>
          completeStage()
        }
        switch.foreach(callback.invoke)
      }

      setHandler(in, new InHandler {
        override def onPush(): Unit = { push(out, grab(in)) }
      })
      setHandler(out, new OutHandler {
        override def onPull(): Unit = { pull(in) }
      })
    }
}
```

Integration with actors

This section is a stub and will be extended in the next release This is an experimental feature*

It is possible to acquire an ActorRef that can be addressed from the outside of the stage, similarly how `AsyncCallback` allows injecting asynchronous events into a stage logic. This reference can be obtained by calling `getStageActorRef(receive)` passing in a function that takes a `Pair` of the sender `ActorRef` and the received message. This reference can be used to watch other actors by calling its `watch(ref)` or `unwatch(ref)` methods. The reference can be also watched by external actors. The current limitations of this `ActorRef` are:

- they are not location transparent, they cannot be accessed via remoting.
- they cannot be returned as materialized values.
- they cannot be accessed from the constructor of the `GraphStageLogic`, but they can be accessed from the `preStart()` method.

Custom materialized values

Custom stages can return materialized values instead of `Unit` by inheriting from `GraphStageWithMaterializedValue` instead of the simpler `GraphStage`. The difference is that in this case the method `createLogicAndMaterializedValue(inheritedAttributes)` needs to be overridden, overridden, and in addition to the stage logic the materialized value must be provided

> **Warning:** There is no built-in synchronization of accessing this value from both of the thread where the logic runs and the thread that got hold of the materialized value. It is the responsibility of the programmer to add the necessary (non-blocking) synchronization and visibility guarantees to this shared object.

In this sample the materialized value is a future containing the first element to go through the stream:

```
class FirstValue[A] extends GraphStageWithMaterializedValue[FlowShape[A, A], Future[A]] {

  val in = Inlet[A]("FirstValue.in")
  val out = Outlet[A]("FirstValue.out")

  val shape = FlowShape.of(in, out)

  override def createLogicAndMaterializedValue(inheritedAttributes: Attributes): (GraphStageLogic,
    val promise = Promise[A]()
    val logic = new GraphStageLogic(shape) {

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          promise.success(elem)
          push(out, elem)

          // replace handler with one just forwarding
          setHandler(in, new InHandler {
            override def onPush(): Unit = {
              push(out, grab(in))
            }
          })
        }
      })

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          pull(in)
        }
      })

    }

    (logic, promise.future)
  }
}
```

Using attributes to affect the behavior of a stage

This section is a stub and will be extended in the next release

Stages can access the `Attributes` object created by the materializer. This contains all the applied (inherited) attributes applying to the stage, ordered from least specific (outermost) towards the most specific (innermost) attribute. It is the responsibility of the stage to decide how to reconcile this inheritance chain to a final effective decision.

See *Modularity, Composition and Hierarchy* for an explanation on how attributes work.

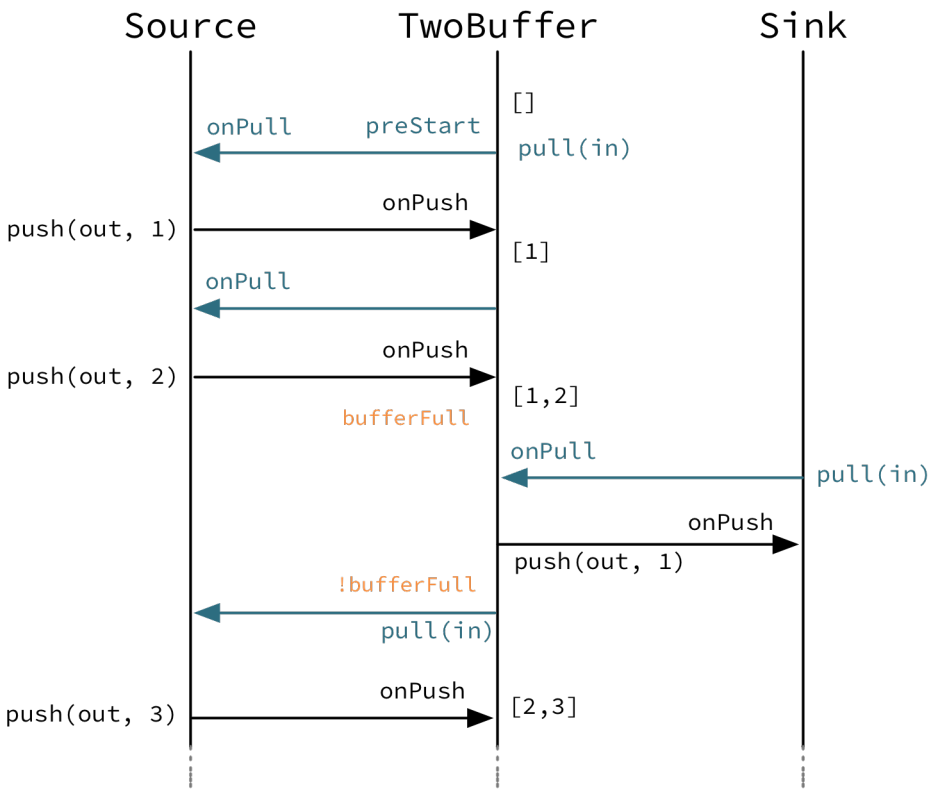
Rate decoupled graph stages

Sometimes it is desirable to *decouple* the rate of the upstream and downstream of a stage, synchronizing only when needed.

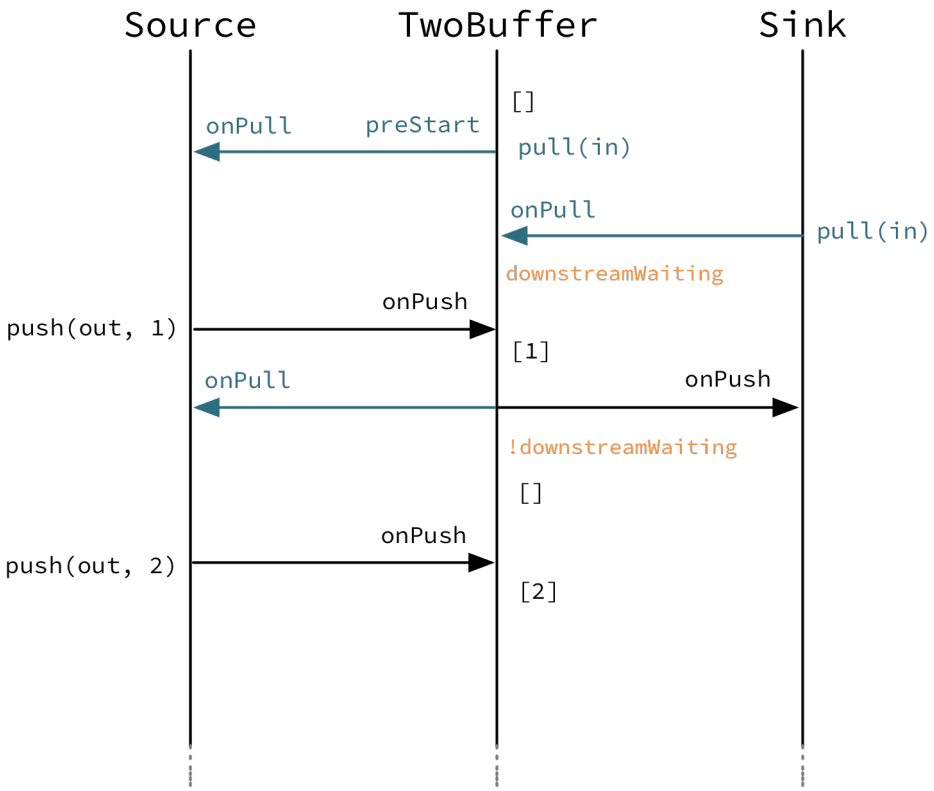
This is achieved in the model by representing a `GraphStage` as a *boundary* between two regions where the demand sent upstream is decoupled from the demand that arrives from downstream. One immediate consequence of this difference is that an `onPush` call does not always lead to calling `push` and an `onPull` call does not always lead to calling `pull`.

One of the important use-case for this is to build buffer-like entities, that allow independent progress of upstream and downstream stages when the buffer is not full or empty, and slowing down the appropriate side if the buffer becomes empty or full.

The next diagram illustrates the event sequence for a buffer with capacity of two elements in a setting where the downstream demand is slow to start and the buffer will fill up with upstream elements before any demand is seen from downstream.



Another scenario would be where the demand from downstream starts coming in before any element is pushed into the buffer stage.



The first difference we can notice is that our `Buffer` stage is automatically pulling its upstream on initialization. The buffer has demand for up to two elements without any downstream demand.

The following code example demonstrates a buffer class corresponding to the message sequence chart above.

```
class TwoBuffer[A] extends GraphStage[FlowShape[A, A]] {

  val in = Inlet[A]("TwoBuffer.in")
  val out = Outlet[A]("TwoBuffer.out")

  val shape = FlowShape.of(in, out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {

      val buffer = mutable.Queue[A]()
      def bufferFull = buffer.size == 2
      var downstreamWaiting = false

      override def preStart(): Unit = {
        // a detached stage needs to start upstream demand
        // itself as it is not triggered by downstream demand
        pull(in)
      }

      setHandler(in, new InHandler {
        override def onPush(): Unit = {
          val elem = grab(in)
          buffer.enqueue(elem)
```

```
          if (downstreamWaiting) {
            downstreamWaiting = false
            val bufferedElem = buffer.dequeue()
            push(out, bufferedElem)
          }
          if (!bufferFull) {
            pull(in)
          }
        }

        override def onUpstreamFinish(): Unit = {
          if (buffer.nonEmpty) {
            // emit the rest if possible
            emitMultiple(out, buffer.toIterator)
          }
          completeStage()
        }
      })

      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          if (buffer.isEmpty) {
            downstreamWaiting = true
          } else {
            val elem = buffer.dequeue
            push(out, elem)
          }
          if (!bufferFull && !hasBeenPulled(in)) {
            pull(in)
          }
        }
      })
    }

}
```

1.8.2 Thread safety of custom processing stages

All of the above custom stages (linear or graph) provide a few simple guarantees that implementors can rely on.

- The callbacks exposed by all of these classes are never called concurrently.
- The state encapsulated by these classes can be safely modified from the provided callbacks, without any further synchronization.

In essence, the above guarantees are similar to what `Actor` s provide, if one thinks of the state of a custom stage as state of an actor, and the callbacks as the `receive` block of the actor.

> **Warning:** It is **not safe** to access the state of any custom stage outside of the callbacks that it provides, just like it is unsafe to access the state of an actor from the outside. This means that Future callbacks should **not close over** internal state of custom stages because such access can be concurrent with the provided callbacks, leading to undefined behavior.

1.9 Integration

1.9.1 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 strategy `OverflowStrategy.backpressure` is not supported for this Source type, you should consider using `ActorPublisher` if you want a backpressured actor interface.

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")
```

A publisher that is created with `Sink.asPublisher` supports a specified number of subscribers. Additional subscription attempts will be rejected with an `IllegalStateException`.

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))
```

1.9.2 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: RunnableGraph[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: RunnableGraph[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: RunnableGraph[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(ActorAttributes.dispatcher("blocking-dispatcher"))
val sendTextMessages: RunnableGraph[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: RunnableGraph[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 materializer = ActorMaterializer(
  ActorMaterializerSettings(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 `ActorMaterializerSettings`.

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 materializer = ActorMaterializer(
  ActorMaterializerSettings(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 `ActorMaterializerSettings`.

1.9.3 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.fromPublisher(tweets).via(authors).to(Sink.fromSubscriber(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 also converted to a `RunnableGraph[Processor[In, Out]]` which materializes to a `Processor` when `run()` is called. `run()` itself can be called multiple times, resulting in a new `Processor` instance each time.

```
val processor: Processor[Tweet, Author] = authors.toProcessor.run()

tweets.subscribe(processor)
processor.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.fromPublisher(tweets).via(authors).runWith(Sink.asPublisher(fanout = false))

authorPublisher.subscribe(storage)
```

A publisher that is created with `Sink.asPublisher(false)` supports only a single subscription. Additional subscription attempts will be rejected with an `IllegalStateException`.

A publisher that supports multiple subscribers using fan-out/broadcasting is created as follows:

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

```
val authorPublisher: Publisher[Author] =
  Source.fromPublisher(tweets).via(authors)
    .runWith(Sink.asPublisher(fanout = true))

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

The input buffer size of the stage 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.fromSubscriber(storage)).runWith(Source.asSubscriber[Tweet])

tweets.subscribe(tweetSubscriber)
```

It is also possible to use re-wrap `Processor` instances as a `Flow` by passing a factory function that will create the `Processor` instances:

```
// An example Processor factory
def createProcessor: Processor[Int, Int] = Flow[Int].toProcessor.run()

val flow: Flow[Int, Int, Unit] = Flow.fromProcessor(() => createProcessor)
```

Please note that a factory is necessary to achieve reusability of the resulting `Flow`.

1.10 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.

Warning: *ZipWith*, *GraphStage* junction, *ActorPublisher* source and *ActorSubscriber* sink components do not honour the supervision strategy attribute yet.

1.10.1 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 materializer = ActorMaterializer()
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 materializer = ActorMaterializer(
  ActorMaterializerSettings(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 materializer = ActorMaterializer()
val decider: Supervision.Decider = {
  case _: ArithmeticException => Supervision.Resume
  case _                      => Supervision.Stop
}
val flow = Flow[Int]
  .filter(100 / _ < 50).map(elem => 100 / (5 - elem))
  .withAttributes(ActorAttributes.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 materializer = ActorMaterializer()
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(ActorAttributes.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))
```

1.10.2 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 ActorAttributes.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`.

1.11 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.

1.11.1 Streaming TCP

Accepting connections: Echo Server

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

```
val binding: Future[ServerBinding] =
  Tcp().bind("127.0.0.1", 8888).to(Sink.ignore).run()

binding.map { b =>
  b.unbind() onComplete {
    case _ => // ...
  }
}
```

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

`Framing.delimiter` helper Flow to chunk the inputs up into actual lines of text. The last boolean argument indicates that we require an explicit line ending even for the last message before the connection is closed. In this example we simply add exclamation marks to each incoming text message and push it through the flow:

```
import akka.stream.io.Framing

val connections: Source[IncomingConnection, Future[ServerBinding]] =
  Tcp().bind(host, port)
connections runForeach { connection =>
  println(s"New connection from: ${connection.remoteAddress}")

  val echo = Flow[ByteString]
    .via(Framing.delimiter(
      ByteString("\n"),
      maximumFrameLength = 256,
      allowTruncation = true))
    .map(_.utf8String)
    .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 a `Sink.cancelled` and its upstream to a `Source.empty`). It is also possible to shut down the server's socket by cancelling the `IncomingConnection` source `connections`.

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]
    .via(Framing.delimiter(
      ByteString("\n"),
      maximumFrameLength = 256,
      allowTruncation = true))
    .map(_.utf8String)
    .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.fromGraph(GraphDSL.create() { implicit b =>
    import GraphDSL.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]
      .via(Framing.delimiter(
        ByteString("\n"),
        maximumFrameLength = 256,
        allowTruncation = true))
      .map(_.utf8String)
      .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)

    FlowShape(echo.in, concat.out)
  })

  connection.handleWith(serverLogic)
}
```

The way we constructed a `Flow` using the `GraphDSL` 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 `Outlet` and exactly one `Inlet` 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 - `BYE` in the case of the server, and `q` in the case of the client. This is implemented by using a custom `PushStage` which completes the stream once it encounters such command.

1.11.2 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 creating a *FileIO.fromFile* 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")

val foreach: Future[Long] = FileIO.fromFile(file)
  .to(Sink.ignore)
  .run()
```

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.stream.blocking-io-dispatcher`, or for a specific stage by specifying a custom Dispatcher in code, like this:

```
FileIO.fromFile(file)
  .withAttributes(ActorAttributes.dispatcher("custom-blocking-io-dispatcher"))
```

1.12 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.

1.12.1 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*.

1.12.2 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.fromGraph(GraphDSL.create() { implicit
  val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
```

[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.

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

  FlowShape(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*

1.12.3 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.fromGraph(GraphDSL.create() { 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)

    FlowShape(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.fromGraph(GraphDSL.create() { 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)

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

val pancakeChefs2: Flow[HalfCookedPancake, Pancake, Unit] =
  Flow.fromGraph(GraphDSL.create() { 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)

    FlowShape(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.

1.13 Testing streams

Verifying behaviour of Akka Stream sources, flows and sinks can be done using various code patterns and libraries. Here we will discuss testing these elements using:

- simple sources, sinks and flows;
- sources and sinks in combination with `TestProbe` from the `akka-testkit` module;
- sources and sinks specifically crafted for writing tests from the `akka-stream-testkit` module.

It is important to keep your data processing pipeline as separate sources, flows and sinks. This makes them easily testable by wiring them up to other sources or sinks, or some test harnesses that `akka-testkit` or `akka-stream-testkit` provide.

1.13.1 Built in sources, sinks and combinators

Testing a custom sink can be as simple as attaching a source that emits elements from a predefined collection, running a constructed test flow and asserting on the results that sink produced. Here is an example of a test for a sink:

```
val sinkUnderTest = Flow[Int].map(_ * 2).toMat(Sink.fold(0)(_ + _))(Keep.right)

val future = Source(1 to 4).runWith(sinkUnderTest)
val result = Await.result(future, 100.millis)
assert(result == 20)
```

The same strategy can be applied for sources as well. In the next example we have a source that produces an infinite stream of elements. Such source can be tested by asserting that first arbitrary number of elements hold some condition. Here the `grouped` combinator and `Sink.head` are very useful.

```
import system.dispatcher
import akka.pattern.pipe

val sourceUnderTest = Source.repeat(1).map(_ * 2)

val future = sourceUnderTest.grouped(10).runWith(Sink.head)
val result = Await.result(future, 100.millis)
assert(result == Seq.fill(10)(2))
```

When testing a flow we need to attach a source and a sink. As both stream ends are under our control, we can choose sources that tests various edge cases of the flow and sinks that ease assertions.

```
val flowUnderTest = Flow[Int].takeWhile(_ < 5)

val future = Source(1 to 10).via(flowUnderTest).runWith(Sink.fold(Seq.empty[Int])(_ :+ _))
val result = Await.result(future, 100.millis)
assert(result == (1 to 4))
```

1.13.2 TestKit

Akka Stream offers integration with Actors out of the box. This support can be used for writing stream tests that use familiar `TestProbe` from the `akka-testkit` API.

One of the more straightforward tests would be to materialize stream to a `Future` and then use `pipe` pattern to pipe the result of that future to the probe.

```
import system.dispatcher
import akka.pattern.pipe

val sourceUnderTest = Source(1 to 4).grouped(2)

val probe = TestProbe()
sourceUnderTest.grouped(2).runWith(Sink.head).pipeTo(probe.ref)
probe.expectMsg(100.millis, Seq(Seq(1, 2), Seq(3, 4)))
```

Instead of materializing to a future, we can use a `Sink.actorRef` that sends all incoming elements to the given `ActorRef`. Now we can use assertion methods on `TestProbe` and expect elements one by one as they arrive. We can also assert stream completion by expecting for `onCompleteMessage` which was given to `Sink.actorRef`.

```
case object Tick
val sourceUnderTest = Source.tick(0.seconds, 200.millis, Tick)

val probe = TestProbe()
val cancellable = sourceUnderTest.to(Sink.actorRef(probe.ref, "completed")).run()

probe.expectMsg(1.second, Tick)
probe.expectNoMsg(100.millis)
probe.expectMsg(200.millis, Tick)
cancellable.cancel()
probe.expectMsg(200.millis, "completed")
```

Similarly to `Sink.actorRef` that provides control over received elements, we can use `Source.actorRef` and have full control over elements to be sent.

```
val sinkUnderTest = Flow[Int].map(_.toString).toMat(Sink.fold("")(_ + _))(Keep.right)

val (ref, future) = Source.actorRef(8, OverflowStrategy.fail)
  .toMat(sinkUnderTest)(Keep.both).run()
```

```
ref ! 1
ref ! 2
ref ! 3
ref ! akka.actor.Status.Success("done")

val result = Await.result(future, 100.millis)
assert(result == "123")
```

1.13.3 Streams TestKit

You may have noticed various code patterns that emerge when testing stream pipelines. Akka Stream has a separate `akka-stream-testkit` module that provides tools specifically for writing stream tests. This module comes with two main components that are `TestSource` and `TestSink` which provide sources and sinks that materialize to probes that allow fluent API.

Note: Be sure to add the module `akka-stream-testkit` to your dependencies.

A sink returned by `TestSink.probe` allows manual control over demand and assertions over elements coming downstream.

```
val sourceUnderTest = Source(1 to 4).filter(_ % 2 == 0).map(_ * 2)

sourceUnderTest
  .runWith(TestSink.probe[Int])
  .request(2)
  .expectNext(4, 8)
  .expectComplete()
```

A source returned by `TestSource.probe` can be used for asserting demand or controlling when stream is completed or ended with an error.

```
val sinkUnderTest = Sink.cancelled

TestSource.probe[Int]
  .toMat(sinkUnderTest)(Keep.left)
  .run()
  .expectCancellation()
```

You can also inject exceptions and test sink behaviour on error conditions.

```
val sinkUnderTest = Sink.head[Int]

val (probe, future) = TestSource.probe[Int]
  .toMat(sinkUnderTest)(Keep.both)
  .run()
probe.sendError(new Exception("boom"))

Await.ready(future, 100.millis)
val Failure(exception) = future.value.get
assert(exception.getMessage == "boom")
```

Test source and sink can be used together in combination when testing flows.

```
val flowUnderTest = Flow[Int].mapAsyncUnordered(2) { sleep =>
  pattern.after(10.millis * sleep, using = system.scheduler)(Future.successful(sleep))
}

val (pub, sub) = TestSource.probe[Int]
  .via(flowUnderTest)
  .toMat(TestSink.probe[Int])(Keep.both)
  .run()
```

```
sub.request(n = 3)
pub.sendNext(3)
pub.sendNext(2)
pub.sendNext(1)
sub.expectNextUnordered(1, 2, 3)

pub.sendError(new Exception("Power surge in the linear subroutine C-47!"))
val ex = sub.expectError()
assert(ex.getMessage.contains("C-47"))
```

1.13.4 Fuzzing Mode

For testing, it is possible to enable a special stream execution mode that exercises concurrent execution paths more aggressively (at the cost of reduced performance) and therefore helps exposing race conditions in tests. To enable this setting add the following line to your configuration:

```
akka.stream.materializer.debug.fuzzing-mode = on
```

Warning: Never use this setting in production or benchmarks. This is a testing tool to provide more coverage of your code during tests, but it reduces the throughput of streams. A warning message will be logged if you have this setting enabled.

1.14 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 supervision 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.

1.14.1 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 *Backpressure aware stages* 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
sliding	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
fold	upstream completes	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
takeWhile	the predicate is true and until the first false result	downstream backpressures	predicate returned false or upstream completes
dropWhile	the predicate returned false and for all following stream elements	predicate returned false and downstream backpressures	upstream completes
recover	the element is available from the upstream or upstream is failed and pf returns an element	downstream backpressures, not when failure happened	upstream completes or upstream failed with exception pf can handle
detach	the upstream stage has emitted and there is demand	downstream backpressures	upstream completes

1.14.2 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).

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]

1.14.3 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

1.14.4 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

1.14.5 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]
splitAfter	an element passes through. When the provided predicate is true it emitts the element * and opens a new substream for subsequent element	there is an element pending for the next substream, but the previous is not fully consumed yet, or the substream backpressures	upstream completes [3]
flatMapConcat	the current consumed substream has an element available	downstream backpressures	upstream completes and all consumed substreams complete
flatMapMerge	one of the currently consumed substreams has an element available	downstream backpressures	upstream completes and all consumed substreams complete

1.14.6 Fan-in stages

Most of these stages can be expressible as a `GraphStage`. 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 (*)
mergeSorted	all of the inputs have 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 have an element available	downstream backpressures	any upstream completes
zipWith	all of the inputs have 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
prepend	the given stream has an element available; if the given input completes, it tries the current one	downstream backpressures	all upstreams complete

(*) This behavior is changeable to completing when any upstream completes by setting `eagerClose=true`.

1.14.7 Fan-out stages

Most of these stages can be expressible as a `GraphStage`. 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.

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

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
unzipWith	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

1.15 Streams Cookbook

1.15.1 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*.

1.15.2 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 (*Buffers and working with rate*, *Working with streaming 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 }
```

Another approach to logging is to use `log()` operation which allows configuring logging for elements flowing through the stream as well as completion and erroring.

```
// customise log levels
mySource.log("before-map")
  .withAttributes(Attributes.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

Situation: 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)` we create a stream of groups with maximum size of `MaxAllowedSeqSize` and then we take the first element of this stream by attaching a `Sink.head`. 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

Situation: 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

Situation: 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.

The `Framing` helper object contains a convenience method to parse messages from a stream of `ByteStrings`:

```
import akka.stream.io.Framing
val linesStream = rawData.via(Framing.delimiter(
  ByteString("\r\n"), maximumFrameLength = 100, allowTruncation = true))
  .map(_.utf8String)
```

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 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). `groupBy` returns a `SubFlow`, which means that we transform the resulting substreams directly. In this case we use the `fold` combinator to aggregate the word itself and the number of its occurrences within a tuple `(String, Integer)`. Each substream will then emit one final value—precisely such a pair—when the overall input completes. As a last step we merge back these values from the substreams into one single output stream.

One noteworthy detail pertains to the `MaximumDistinctWords` parameter: this defines the breadth of the groupBy and merge operations. Akka Streams is focused on bounded resource consumption and the number of concurrently open inputs to the merge operator describes the amount of resources needed by the merge itself. Therefore only a finite number of substreams can be active at any given time. If the `groupBy` operator encounters more keys than this number then the stream cannot continue without violating its resource bound, in this case `groupBy` will terminate with a failure.

```
val counts: Source[(String, Int), Unit] = words
  // split the words into separate streams first
  .groupBy(MaximumDistinctWords, identity)
  // add counting logic to the streams
  .fold(("", 0)) {
    case ((_, count), word) => (word, count + 1)
  }
  // get a stream of word counts
  .mergeSubstreams
```

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] = {

  Flow[In]
    .groupBy(maximumGroupSize, groupKey)
    .fold(Option.empty[(K, Out)]) {
      case (None, elem) =>
        val key = groupKey(elem)
        Some((key, fold(foldZero(key), elem)))
      case (Some((key, out)), elem) =>
        Some((key, fold(out, elem)))
    }
    .map(_.get)
    .mergeSubstreams
}

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 reading the overall input stream, 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] = extractTopics

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 = messageAndTopic
  .groupBy(2, _._2).map {
    case (msg, topic) =>
      // do what needs to be done
  }
```

1.15.3 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 = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.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
  ClosedShape
})
```

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 = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._
  val zip = builder.add(ZipWith((msg: Message, trigger: Trigger) => msg))

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

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.

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

  Flow.fromGraph(GraphDSL.create() { 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
    }

    FlowShape(balancer.in, merge.out)
  })
}
```

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

1.15.4 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 = RunnableGraph.fromGraph(GraphDSL.create(mySink1, mySink2, mySink3)((_, _, _)) { implic
  (sink1, sink2, sink3) =>
    import GraphDSL.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
    ClosedShape
})
```

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 flow 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)

```
val missedTicks: Flow[Tick, Int, Unit] =
  Flow[Tick].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` or 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 `holdDownstream()` 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 `holdDownstream()` 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 =
    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.

1.15.5 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)

  override def onUpstreamFinish(ctx: Context[ByteString]): TerminationDirective =
    if (buffer.nonEmpty) ctx.absorbTermination()
    else ctx.finish()

  private def emitChunkOrPull(ctx: Context[ByteString]): SyncDirective = {
    if (buffer.isEmpty) {
      if (ctx.isFinishing) ctx.finish()
      else 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.

There is a built-in operation that allows to do this directly:

```
import scala.concurrent.duration._
val injectKeepAlive: Flow[ByteString, ByteString, Unit] =
  Flow[ByteString].keepAlive(1.second, () => keepaliveMessage)
```

1.16 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

      # Enable automatic fusing of all graphs that are run. For short-lived streams
      # this may cause an initial runtime overhead, but most of the time fusing is
      # desirable since it reduces the number of Actors that are created.
      auto-fusing = on

      debug {
        # Enables the fuzzing mode which increases the chance of race conditions
        # by aggressively reordering events and making certain operations more
        # concurrent than usual.
        # This setting is for testing purposes, NEVER enable this in a production
        # environment!
        # To get the best results, try combining this setting with a throughput
        # of 1 on the corresponding dispatchers.
        fuzzing-mode = off
      }
    }

    # 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.
    blocking-io-dispatcher = "akka.stream.default-blocking-io-dispatcher"

    default-blocking-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
      }
    }
  }

  # configure overrides to ssl-configuration here (to be used by akka-streams, and akka-http - i.e
  ssl-config {
    # due to still supporting JDK6 in this release
    # TODO once JDK 8 is required switch this to TLSv1.2 (or remove entirely, leave up to ssl-conf
    protocol = "TLSv1"
  }
}
```

1.17 Migration Guide 1.0 to 2.x

The 2.0 release contains some structural changes that require some simple, mechanical source-level changes in client code. While these are detailed below, there is another change that may have an impact on the runtime behavior of your streams and which therefore is listed first.

1.17.1 Operator Fusion is on by default

Akka Streams 2.0 contains an initial version of stream operator fusion support. This means that the processing steps of a flow or stream graph can be executed within the same Actor and has three consequences:

- starting up a stream may take longer than before due to executing the fusion algorithm
- passing elements from one processing stage to the next is a lot faster between fused stages due to avoiding the asynchronous messaging overhead
- fused stream processing stages do no longer run in parallel to each other, meaning that only up to one CPU core is used for each fused part

The first point can be countered by pre-fusing and then reusing a stream blueprint, see `akka.stream.Fusing`. In order to balance the effects of the second and third bullet points you will have to insert asynchronous boundaries manually into your flows and graphs by way of adding `Attributes.asyncBoundary` to pieces that shall communicate with the rest of the graph in an asynchronous fashion.

Warning: Without fusing (i.e. up to version 2.0-M2) each stream processing stage had an implicit input buffer that holds a few elements for efficiency reasons. If your flow graphs contain cycles then these buffers may have been crucial in order to avoid deadlocks. With fusing these implicit buffers are no longer there, data elements are passed without buffering between fused stages. In those cases where buffering is needed in order to allow the stream to run at all, you will have to insert explicit buffers with the `.buffer()` combinator—typically a buffer of size 2 is enough to allow a feedback loop to function.

The new fusing behavior can be disabled by setting the configuration parameter `akka.stream.materializer.auto-fusing=off`. In that case you can still manually fuse those graphs which shall run on less Actors. Fusable elements are

- all GraphStages (this includes all built-in junctions apart from `groupBy`)
- all Stages (this includes all built-in linear operators)
- TCP connections

1.17.2 Introduced proper named constructor methods insted of `wrap()`

There were several, unrelated uses of `wrap()` which made it hard to find and hard to understand the intention of the call. Therefore these use-cases now have methods with different names, helping Java 8 type inference (by reducing the number of overloads) and finding relevant methods in the documentation.

Creating a Flow from other stages

It was possible to create a `Flow` from a graph with the correct shape (`FlowShape`) using `wrap()`. Now this must be done with the more descriptive method `Flow.fromGraph()`.

It was possible to create a `Flow` from a `Source` and a `Sink` using `wrap()`. Now this functionality can be accessed trough the more descriptive methods `Flow.fromSinkAndSource` and `Flow.fromSinkAndSourceMat`.

Creating a BidiFlow from other stages

It was possible to create a `BidiFlow` from a graph with the correct shape (`BidiShape`) using `wrap()`. Now this must be done with the more descriptive method `BidiFlow.fromGraph()`.

It was possible to create a `BidiFlow` from two `Flow` s using `wrap()`. Now this functionality can be accessed trough the more descriptive methods `BidiFlow.fromFlows` and `BidiFlow.fromFlowsMat`.

It was possible to create a `BidiFlow` from two functions using `apply()` (Scala DSL) or `create()` (Java DSL). Now this functionality can be accessed trough the more descriptive method `BidiFlow.fromFunctions`.

Update procedure

1. Replace all uses of `Flow.wrap` when it converts a `Graph` to a `Flow` with `Flow.fromGraph`
2. Replace all uses of `Flow.wrap` when it converts a `Source` and `Sink` to a `Flow` with `Flow.fromSinkAndSource` or `Flow.fromSinkAndSourceMat`
3. Replace all uses of `BidiFlow.wrap` when it converts a `Graph` to a `BidiFlow` with `BidiFlow.fromGraph`
4. Replace all uses of `BidiFlow.wrap` when it converts two `Flow` s to a `BidiFlow` with `BidiFlow.fromFlows` or `BidiFlow.fromFlowsMat`
5. Replace all uses of `BidiFlow.apply()` when it converts two functions to a `BidiFlow` with `BidiFlow.fromFunctions`

Example

```
val graphSource: Graph[SourceShape[Int], Unit] = ???
// This no longer works!
val source: Source[Int, Unit] = Source.wrap(graphSource)

val graphSink: Graph[SinkShape[Int], Unit] = ???
// This no longer works!
val sink: Sink[Int, Unit] = Sink.wrap(graphSink)

val graphFlow: Graph[FlowShape[Int, Int], Unit] = ???
// This no longer works!
val flow: Flow[Int, Int, Unit] = Flow.wrap(graphFlow)

// This no longer works
Flow.wrap(Sink.head[Int], Source.single(0))(Keep.left)
```

should be replaced by

```
val graphSource: Graph[SourceShape[Int], Unit] = ???
val source: Source[Int, Unit] = Source.fromGraph(graphSource)

val graphSink: Graph[SinkShape[Int], Unit] = ???
val sink: Sink[Int, Unit] = Sink.fromGraph(graphSink)

val graphFlow: Graph[FlowShape[Int, Int], Unit] = ???
val flow: Flow[Int, Int, Unit] = Flow.fromGraph(graphFlow)

Flow.fromSinkAndSource(Sink.head[Int], Source.single(0))
```

and

```
val bidiGraph: Graph[BidiShape[Int, Int, Int, Int], Unit = ???
// This no longer works!
val bidi: BidiFlow[Int, Int, Int, Int, Unit] = BidiFlow.wrap(bidiGraph)

// This no longer works!
BidiFlow.wrap(flow1, flow2)(Keep.both)

// This no longer works!
BidiFlow((x: Int) => x + 1, (y: Int) => y * 3)
```

Should be replaced by

```
val bidiGraph: Graph[BidiShape[Int, Int, Int, Int], Unit] = ???
val bidi: BidiFlow[Int, Int, Int, Int, Unit] = BidiFlow.fromGraph(bidiGraph)
```

```
BidiFlow.fromFlows(flow1, flow2)

BidiFlow.fromFunctions((x: Int) => x + 1, (y: Int) => y * 3)
```

1.17.3 FlowGraph class and builder methods have been renamed

Due to incorrect overlap with the `Flow` concept we renamed the `FlowGraph` class to `GraphDSL`. There is now only one graph creation method called `create` which is analogous to the old `partial` method. For closed graphs now it is explicitly required to return `ClosedShape` at the end of the builder block.

Update procedure

1. Search and replace all occurrences of `FlowGraph` with `GraphDSL`.
2. Replace all occurrences of `GraphDSL.partial()` or `GraphDSL.closed()` with `GraphDSL.create()`.
3. Add `ClosedShape` as a return value of the builder block if it was `FlowGraph.closed()` before.
4. Wrap the closed graph with `RunnableGraph.fromGraph` if it was `FlowGraph.closed()` before.

Example

```
// This no longer works!
FlowGraph.closed() { builder =>
  //...
}

// This no longer works!
FlowGraph.partial() { builder =>
  //...
  FlowShape(inlet, outlet)
}
```

should be replaced by

```
// Replaces GraphDSL.closed()
GraphDSL.create() { builder =>
  //...
  ClosedShape
}

// Replaces GraphDSL.partial()
GraphDSL.create() { builder =>
  //...
  FlowShape(inlet, outlet)
}
```

1.17.4 Methods that create Source, Sink, Flow from Graphs have been removed

Previously there were convenience methods available on `Sink`, `Source`, `Flow` an `BidiFlow` to create these DSL elements from a graph builder directly. Now this requires two explicit steps to reduce the number of overloaded methods (helps Java 8 type inference) and also reduces the ways how these elements can be created. There is only one graph creation method to learn (`GraphDSL.create`) and then there is only one conversion method to use `fromGraph()`.

This means that the following methods have been removed:

- `adapt()` method on `Source`, `Sink`, `Flow` and `BidiFlow` (both DSLs)

- `apply()` overloads providing a graph `Builder` on `Source`, `Sink`, `Flow` and `BidiFlow` (Scala DSL)
- `create()` overloads providing a graph `Builder` on `Source`, `Sink`, `Flow` and `BidiFlow` (Java DSL)

Update procedure

Everywhere where `Source`, `Sink`, `Flow` and `BidiFlow` is created from a graph using a builder have to be replaced with two steps

1. Create a `Graph` with the correct `Shape` using `GraphDSL.create` (e.g.. for `Source` it means first creating a `Graph` with `SourceShape`)
2. Create the required DSL element by calling `fromGraph()` on the required DSL element (e.g. `Source.fromGraph`) passing the graph created in the previous step

Example

```
// This no longer works!
Source() { builder =>
  //...
  outlet
}

// This no longer works!
Sink() { builder =>
  //...
  inlet
}

// This no longer works!
Flow() { builder =>
  //...
  (inlet, outlet)
}

// This no longer works!
BidiFlow() { builder =>
  //...
  BidiShape(inlet1, outlet1, inlet2, outlet2)
}
```

should be replaced by

```
Source.fromGraph(
  GraphDSL.create() { builder =>
    //...
    SourceShape(outlet)
  })

Sink.fromGraph(
  GraphDSL.create() { builder =>
    //...
    SinkShape(inlet)
  })

Flow.fromGraph(
  GraphDSL.create() { builder =>
    //...
    FlowShape(inlet, outlet)
  })
```

```
BidiFlow.fromGraph(
  GraphDSL.create() { builder =>
    //...
    BidiShape(inlet1, outlet1, inlet2, outlet2)
  })
```

1.17.5 Several Graph builder methods have been removed

The `addEdge` methods have been removed from the DSL to reduce the ways connections can be made and to reduce the number of overloads. Now only the `~>` notation is available which requires the import of the implicits `GraphDSL.Implicits._`.

Update procedure

1. Replace all uses of `scaladsl.Builder.addEdge(Outlet, Inlet)` by the graphical DSL `~>`.
2. Replace all uses of `scaladsl.Builder.addEdge(Outlet, FlowShape, Inlet)` by the graphical DSL `~>`. methods, or the graphical DSL `~>`.
3. Import `FlowGraph.Implicits._` in the builder block or an enclosing scope.

Example

```
FlowGraph.closed() { builder =>
  //...
  // This no longer works!
  builder.addEdge(outlet, inlet)
  // This no longer works!
  builder.addEdge(outlet, flow1, inlet)
  //...
}
```

should be replaced by

```
RunnableGraph.fromGraph(
  GraphDSL.create() { implicit builder =>
    import GraphDSL.Implicits._
    outlet ~> inlet
    outlet ~> flow ~> inlet
    //...
    ClosedShape
  })
```

1.17.6 Source constructor name changes

`Source.lazyEmpty` has been replaced by `Source.maybe` which returns a `Promise` that can be completed by one or zero elements by providing an `Option`. This is different from `lazyEmpty` which only allowed completion to be sent, but no elements.

The `apply()` overload on `Source` has been refactored to separate methods to reduce the number of overloads and make source creation more discoverable.

`Source.subscriber` has been renamed to `Source.asSubscriber`.

Update procedure

1. All uses of `Source.lazyEmpty` should be replaced by `Source.maybe` and the returned `Promise` completed with a `None` (an empty `Option`)
2. Replace all uses of `Source(delay,interval,tick)` with the method `Source.tick(delay,interval,tick)`
3. Replace all uses of `Source(publisher)` with the method `Source.fromPublisher(publisher)`
4. Replace all uses of `Source(() => iterator)` with the method `Source.fromIterator(() => iterator))`
5. Replace all uses of `Source(future)` with the method `Source.fromFuture(future))`
6. Replace all uses of `Source.subscriber` with the method `Source.asSubscriber`

Example

```
// This no longer works!
val src: Source[Int, Promise[Unit]] = Source.lazyEmpty[Int]
//...
promise.trySuccess(())

// This no longer works!
val ticks = Source(1.second, 3.seconds, "tick")

// This no longer works!
val pubSource = Source(TestPublisher.manualProbe[Int]())

// This no longer works!
val itSource = Source(() => Iterator.continually(Random.nextGaussian))

// This no longer works!
val futSource = Source(Future.successful(42))

// This no longer works!
val subSource = Source.subscriber
```

should be replaced by

```
val src: Source[Int, Promise[Option[Int]]] = Source.maybe[Int]
//...
// This finishes the stream without emitting anything, just like Source.lazyEmpty did
promise.trySuccess(Some(()))

val ticks = Source.tick(1.second, 3.seconds, "tick")

val pubSource = Source.fromPublisher(TestPublisher.manualProbe[Int]())

val itSource = Source.fromIterator(() => Iterator.continually(Random.nextGaussian))

val futSource = Source.fromFuture(Future.successful(42))

val subSource = Source.asSubscriber
```

1.17.7 Sink constructor name changes

`Sink.apply(subscriber)` has been renamed to `Sink.fromSubscriber(subscriber)` to reduce the number of overloads and make sink creation more discoverable.

Update procedure

1. Replace all uses of `Sink(subscriber)` with the method `Sink.fromSubscriber(subscriber)`

Example

```
// This no longer works!
val subSink = Sink(TestSubscriber.manualProbe[Int]())
```

should be replaced by

```
val subSink = Sink.fromSubscriber(TestSubscriber.manualProbe[Int]())
```

1.17.8 `flatten(FlattenStrategy)` has been replaced by named counterparts

To simplify type inference in Java 8 and to make the method more discoverable, `flatten(FlattenStrategy.concat)` has been removed and replaced with the alternative method `flatten(FlattenStrategy.concat)`.

Update procedure

1. Replace all occurrences of `flatten(FlattenStrategy.concat)` with `flatMapConcat(identity)`
2. Consider replacing all occurrences of `map(f).flatMapConcat(identity)` with `flatMapConcat(f)`

Example

```
// This no longer works!
Flow[Source[Int, Any]].flatten(FlattenStrategy.concat)
```

should be replaced by

```
Flow[Source[Int, Any]].flatMapConcat(identity)
```

1.17.9 *Sink.fanoutPublisher() and Sink.publisher() is now a single method*

It was a common user mistake to use `Sink.publisher` and get into trouble since it would only support a single `Subscriber`, and the discoverability of the apprpriate fix was non-obvious (Sink.fanoutPublisher). To make the decision whether to support fanout or not an active one, the aforementioned methods have been replaced with a single method: `Sink.asPublisher(fanout: Boolean)`.

Update procedure

1. Replace all occurences of `Sink.publisher` with `Sink.asPublisher(false)`
2. Replace all occurences of `Sink.fanoutPublisher` with `Sink.asPublisher(true)`

Example

```
// This no longer works!
val subSink = Sink.publisher

// This no longer works!
val subSink = Sink.fanoutPublisher(2, 8)
```

should be replaced by

```
val pubSink = Sink.asPublisher(fanout = false)

val pubSinkFanout = Sink.asPublisher(fanout = true)
```

1.17.10 FlexiMerge an FlexiRoute has been replaced by GraphStage

The `FlexiMerge` and `FlexiRoute` DSLs have been removed since they provided an abstraction that was too limiting and a better abstraction have been created which is called `GraphStage`. `GraphStage` can express fan-in and fan-out stages, but many other constructs as well with possibly multiple input and output ports (e.g. a `BidiStage`).

This new abstraction provides a more uniform way to crate custom stream processing stages of arbitrary `Shape`. In fact, all of the built-in fan-in and fan-out stages are now implemented in terms of `GraphStage`.

Update procedure

There is no simple update procedure. The affected stages must be ported to the new ''GraphStage'' DSL manually. Please read the `GraphStage` *documentation (TODO) for details.*

1.17.11 GroupBy, SplitWhen and SplitAfter now return SubFlow

Previously the `groupBy`, `splitWhen`, and `splitAfter` combinators returned a type that included a `Source` within its elements. Transforming these substreams was only possible by nesting the respective combinators inside a `map` of the outer stream. This has been made more convenient and also safer by dropping down into transforming the substreams instead: the return type is now a `SubFlow` that does not implement the `Graph` interface and therefore only represents an unfinished intermediate builder step. The substream mode can be ended by closing the substreams (i.e. attaching a `Sink`) or merging them back together.

Update Procedure

The transformations that were done on the substreams need to be lifted up one level. This only works for cases where the processing topology is homogenous for all substreams.

Example

```
Flow[Int]
  // This no longer works!
  .groupBy(_ % 2)
  // This no longer works!
  .map {
    case (key, source) => source.map(_ + 3)
  }
  // This no longer works!
  .flatten(FlattenStrategy.concat)
```

This is implemented now as

```
Flow[Int]
  .groupBy(2, _ % 2) // the first parameter sets max number of substreams
  .map(_ + 3)
  .concatSubstreams
```

Example 2

```
Flow[String]
  // This no longer works!
  .groupBy(identity)
  // This no longer works!
  .map {
    case (key, source) => source.runFold((key, 0))((pair, word) => (key, pair._2 + 1))
  }
  // This no longer works!
  .mapAsyncUnordered(4, identity)
```

This is implemented now as

```
Flow[String]
  .groupBy(MaxDistinctWords, identity)
  .fold(("", 0))((pair, word) => (word, pair._2 + 1))
  .mergeSubstreams
```

1.17.12 Variance of Inlet and Outlet

Scala uses *declaration site variance* which was cumbersome in the cases of `Inlet` and `Outlet` as they are purely symbolic object containing no fields or methods and which are used both in input and output locations (wiring an `Outlet` into an `Inlet`; reading in a stage from an `Inlet`). Because of this reasons all users of these port abstractions now use *use-site variance* (just like Java variance works). This in general does not affect user code expect the case of custom shapes, which now require `@uncheckedVariance` annotations on their `Inlet` and `Outlet` members (since these are now invariant, but the Scala compiler does not know that they have no fields or methods that would violate variance constraints)

This change does not affect Java DSL users.

Update procedure

1. All custom shapes must use `@uncheckedVariance` on their `Inlet` and `Outlet` members.

1.17.13 Renamed `inlet()` and `outlet()` to `in()` and `out()` in `SourceShape`, `SinkShape` and `FlowShape`

The input and output ports of these shapes where called `inlet()` and `outlet()` compared to other shapes that consistently used `in()` and `out()`. Now all `Shape`s use `in()` and `out()`.

Update procedure

Change all references to `inlet()` to `in()` and all references to `outlet()` to `out()` when referring to the ports of `FlowShape`, `SourceShape` and `SinkShape`.

1.17.14 Semantic change in `isHoldingUpstream` in the DetachedStage DSL

The `isHoldingUpstream` method used to return true if the upstream port was in holding state and a completion arrived (inside the `onUpstreamFinished` callback). Now it returns `false` when the upstream is completed.

Update procedure

1. Those stages that relied on the previous behavior need to introduce an extra `Boolean` field with initial value `false`
2. This field must be set on every call to `holdUpstream()` (and variants).
3. In completion, instead of calling `isHoldingUpstream` read this variable instead.

See the example in the AsyncStage migration section for an example of this procedure.

1.17.15 StatefulStage has been replaced by GraphStage

The `StatefulStage` class had some flaws and limitations, most notably around completion handling which caused subtle bugs. The new `GraphStage` (*graphstage-java*) solves these issues and should be used instead.

Update procedure

There is no mechanical update procedure available. Please consult the `GraphStage` documentation (*graphstage-java*).

1.17.16 AsyncStage has been replaced by GraphStage

Due to its complexity and inflexibility `AsyncStage` have been removed in favor of `GraphStage`. Existing `AsyncStage` implementations can be ported in a mostly mechanical way.

Update procedure

1. The subclass of `AsyncStage` should be replaced by `GraphStage`
2. The new subclass must define an `in` and `out` port (`Inlet` and `Outlet` instance) and override the `shape` method returning a `FlowShape`
3. An instance of `GraphStageLogic` must be returned by overriding `createLogic()`. The original processing logic and state will be encapsulated in this `GraphStageLogic`
4. Using `setHandler(port, handler)` and `InHandler` instance should be set on `in` and an `OutHandler` should be set on `out`
5. `onPush`, `onUpstreamFinished` and `onUpstreamFailed` are now available in the `InHandler` subclass created by the user
6. `onPull` and `onDownstreamFinished` are now available in the `OutHandler` subclass created by the user
7. the callbacks above no longer take an extra *ctxt* context parameter.
8. `onPull` only signals the stage, the actual element can be obtained by calling `grab(in)`
9. `ctx.push(elem)` is now `push(out, elem)`
10. `ctx.pull()` is now `pull(in)`
11. `ctx.finish()` is now `completeStage()`
12. `ctx.pushAndFinish(elem)` is now simply two calls: `push(out, elem)`; `completeStage()`
13. `ctx.fail(cause)` is now `failStage(cause)`
14. `ctx.isFinishing()` is now `isClosed(in)`

15. `ctx.absorbTermination()` can be replaced with `if (isAvailable(shape.outlet)) <call the onPull() handler>`
16. `ctx.pushAndPull(elem)` can be replaced with `push(out, elem); pull(in)`
17. `ctx.holdUpstreamAndPush` and `context.holdDownstreamAndPull` can be replaced by simply `push(elem)` and `pull()` respectively
18. The following calls should be removed: `ctx.ignore()`, `ctx.holdUpstream()` and `ctx.holdDownstream()`.
19. `ctx.isHoldingUpstream()` can be replaced with `isAvailable(out)`
20. `ctx.isHoldingDowntream()` can be replaced with `!(isClosed(in) || hasBeenPulled(in))`
21. `ctx.getAsyncCallback()` is now `getAsyncCallback(callback)` which now takes a callback as a parameter. This would correspond to the `onAsyncInput()` callback in the original `AsyncStage`

We show the necessary steps in terms of an example `AsyncStage`

Example

```
class MapAsyncOne[In, Out](f: In ⇒ Future[Out])(implicit ec: ExecutionContext)
  extends AsyncStage[In, Out, Try[Out]] {

  private var elemInFlight: Out = _

  override def onPush(elem: In, ctx: AsyncContext[Out, Try[Out]]) = {
    val future = f(elem)
    val cb = ctx.getAsyncCallback
    future.onComplete(cb.invoke)
    ctx.holdUpstream()
  }

  override def onPull(ctx: AsyncContext[Out, Try[Out]]) =
    if (elemInFlight != null) {
      val e = elemInFlight
      elemInFlight = null.asInstanceOf[Out]
      pushIt(e, ctx)
    } else ctx.holdDownstream()

  override def onAsyncInput(input: Try[Out], ctx: AsyncContext[Out, Try[Out]]) =
    input match {
      case Failure(ex)                          ⇒ ctx.fail(ex)
      case Success(e) if ctx.isHoldingDownstream ⇒ pushIt(e, ctx)
      case Success(e) ⇒
        elemInFlight = e
        ctx.ignore()
    }

  override def onUpstreamFinish(ctx: AsyncContext[Out, Try[Out]]) =
    if (ctx.isHoldingUpstream) ctx.absorbTermination()
    else ctx.finish()

  private def pushIt(elem: Out, ctx: AsyncContext[Out, Try[Out]]) =
    if (ctx.isFinishing) ctx.pushAndFinish(elem)
    else ctx.pushAndPull(elem)
}
```

should be replaced by

```
class MapAsyncOne[In, Out](f: In ⇒ Future[Out])(implicit ec: ExecutionContext)
  extends GraphStage[FlowShape[In, Out]] {
```

```
val in: Inlet[In] = Inlet("MapAsyncOne.in")
val out: Outlet[Out] = Outlet("MapAsyncOne.out")
override val shape: FlowShape[In, Out] = FlowShape(in, out)

// The actual logic is encapsulated in a GraphStageLogic now
override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
  new GraphStageLogic(shape) {

    // All of the state *must* be encapsulated in the GraphStageLogic,
    // not in the GraphStage
    private var elemInFlight: Out = _

    val callback = getAsyncCallback(onAsyncInput)
    var holdingUpstream = false

    // All upstream related events now are handled in an InHandler instance
    setHandler(in, new InHandler {
      // No context or element parameter for onPush
      override def onPush(): Unit = {
        // The element is not passed as an argument but needs to be dequeued explicitly
        val elem = grab(in)
        val future = f(elem)
        future.onComplete(callback.invoke)
        // ctx.holdUpstream is no longer needed, but we need to track the state
        holdingUpstream = true
      }

      // No context parameter
      override def onUpstreamFinish(): Unit = {
        if (holdingUpstream) absorbTermination()
        else completeStage() // ctx.finish turns into completeStage()
      }
    })

    setHandler(out, new OutHandler {
      override def onPull(): Unit = {
        if (elemInFlight != null) {
          val e = elemInFlight
          elemInFlight = null.asInstanceOf[Out]
          pushIt(e)
        } // holdDownstream is no longer needed
      }
    })

    // absorbTermination turns into the code below.
    // This emulates the behavior of the AsyncStage stage.
    private def absorbTermination(): Unit =
      if (isAvailable(shape.out)) getHandler(out).onPull()

    // The line below emulates the behavior of the AsyncStage holdingDownstream
    private def holdingDownstream(): Boolean =
      !(isClosed(in) || hasBeenPulled(in))

    // Any method can be used as a callback, we chose the previous name for
    // easier comparison with the original code
    private def onAsyncInput(input: Try[Out]) =
      input match {
        case Failure(ex)                       ⇒ failStage(ex)
        case Success(e) if holdingDownstream() ⇒ pushIt(e)
        case Success(e) ⇒
          elemInFlight = e
        // ctx.ignore is no longer needed
      }
```

```
      private def pushIt(elem: Out): Unit = {
        // ctx.isFinishing turns into isClosed(in)
        if (isClosed(in)) {
          // pushAndFinish is now two actions
          push(out, elem)
          completeStage()
        } else {
          // pushAndPull is now two actions
          push(out, elem)
          pull(in)
          holdingUpstream = false
        }
      }
    }

}
```

1.17.17 Akka HTTP: Uri parsing mode relaxed-with-raw-query replaced with rawQueryString

Previously Akka HTTP allowed to configure the parsing mode of an Uri's Query part (`?a=b&c=d`) to `relaxed-with-raw-query` which is useful when Uris are not formatted using the usual "key/value pairs" syntax.

Instead of exposing it as an option for the parser, this is now available as the `rawQueryString(): Option[String]` / `queryString(): Option[String]` methods on on `model.Uri`.

For parsing the Query part use `query(charset: Charset = UTF8, mode: Uri.ParsingMode = Uri.ParsingMode.Relaxed): Query`.

Update procedure

1. If the `uri-parsing-mode` was set to `relaxed-with-raw-query`, remove it
2. In places where the query string was accessed in `relaxed-with-raw-query` mode, use the `rawQueryString`/`queryString` methods instead
3. In places where the parsed query parts (such as `parameter`) were used, invoke parsing directly using `uri.query().get("a")`

Example

```
// config, no longer works
akka.http.parsing.uri-parsing-mode = relaxed-with-raw-query
```

should be replaced by:

```
val queryPart: Option[String] = uri.rawQueryString
```

And use of query parameters from `Uri` that looked like this:

```
// This no longer works!
uri.parameter("name")
```

should be replaced by:

```
val param: Option[String] = uri.query().get("a")
```

1.17.18 SynchronousFileSource and SynchronousFileSink

`SynchronousFileSource` and `SynchronousFileSink` have been replaced by `FileIO.read(...)` and `FileIO.write(...)` due to discoverability issues paired with names which leaked internal implementation details.

Update procedure

Replace `SynchronousFileSource(` and `SynchronousFileSource.apply(` with `FileIO.fromFile(`

Replace `SynchronousFileSink(` and `SynchronousFileSink.apply(` with `FileIO.toFile(`

Example

```
// This no longer works!
val fileSrc = SynchronousFileSource(new File("."))

// This no longer works!
val otherFileSrc = SynchronousFileSource(new File("."), 1024)

// This no longer works!
val someFileSink = SynchronousFileSink(new File("."))
```

should be replaced by

```
val fileSrc = FileIO.fromFile(new File("."))

val otherFileSrc = FileIO.fromFile(new File("."), 1024)

val someFileSink = FileIO.toFile(new File("."))
```

1.17.19 InputStreamSource and OutputStreamSink

Both have been replaced by `StreamConverters.fromInputStream(...)` and `StreamConverters.fromOutputStream(...)` due to discoverability issues.

Update procedure

Replace `InputStreamSource(` and `InputStreamSource.apply(` with `StreamConverters.fromInputStream(` i Replace `OutputStreamSink(` and `OutputStreamSink.apply(` with `StreamConverters.fromOutputStream(`

Example

```
// This no longer works!
val inputStreamSrc = InputStreamSource(() => new SomeInputStream())

// This no longer works!
val otherInputStreamSrc = InputStreamSource(() => new SomeInputStream(), 1024)

// This no longer works!
val someOutputStreamSink = OutputStreamSink(() => new SomeOutputStream())
```

should be replaced by

```
val inputStreamSrc = StreamConverters.fromInputStream(() => new SomeInputStream())

val otherInputStreamSrc = StreamConverters.fromInputStream(() => new SomeInputStream())

val someOutputStreamSink = StreamConverters.fromOutputStream(() => new SomeOutputStream())
```

1.17.20 OutputStreamSource and InputStreamSink

Both have been replaced by `StreamConverters.asOutputStream(...)` and `StreamConverters.asInputStream(...)` due to discoverability issues.

Update procedure

Replace `OutputStreamSource(` and `OutputStreamSource.apply(` with `StreamConverters.asOutputStream(`

Replace `InputStreamSink(` and `InputStreamSink.apply(` with `StreamConverters.asInputStream(`

Example

```
// This no longer works!
val outputStreamSrc = OutputStreamSource()

// This no longer works!
val otherOutputStreamSrc = OutputStreamSource(timeout)

// This no longer works!
val someInputStreamSink = InputStreamSink()

// This no longer works!
val someOtherInputStreamSink = InputStreamSink(timeout);
```

should be replaced by

```
val timeout: FiniteDuration = 0.seconds

val outputStreamSrc = StreamConverters.asOutputStream()

val otherOutputStreamSrc = StreamConverters.asOutputStream(timeout)

val someInputStreamSink = StreamConverters.asInputStream()

val someOtherInputStreamSink = StreamConverters.asInputStream(timeout)
```

CHAPTER

TWO

AKKA HTTP

2.1 Introduction

The Akka HTTP modules implement a full server- and client-side HTTP stack on top of *akka-actor* and *akka-stream*. It's not a web-framework but rather a more general toolkit for providing and consuming HTTP-based services. While interaction with a browser is of course also in scope it is not the primary focus of Akka HTTP.

Akka HTTP follows a rather open design and many times offers several different API levels for "doing the same thing". You get to pick the API level of abstraction that is most suitable for your application. This means that, if you have trouble achieving something using a high-level API, there's a good chance that you can get it done with a low-level API, which offers more flexibility but might require you to write more application code.

Akka HTTP is structured into several modules:

akka-http-core A complete, mostly low-level, server- and client-side implementation of HTTP (incl. WebSockets)

akka-http Higher-level functionality, like (un)marshalling, (de)compression as well as a powerful DSL for defining HTTP-based APIs on the server-side

akka-http-testkit A test harness and set of utilities for verifying server-side service implementations

akka-http-spray-json Predefined glue-code for (de)serializing custom types from/to JSON with spray-json

akka-http-xml Predefined glue-code for (de)serializing custom types from/to XML with scala-xml

2.2 Configuration

Just like any other Akka module Akka HTTP is configured via Typesafe Config. Usually this means that you provide an `application.conf` which contains all the application-specific settings that differ from the default ones provided by the reference configuration files from the individual Akka modules.

These are the relevant default configuration values for the Akka HTTP modules.

2.2.1 akka-http-core

```
##########################################
# akka-http-core Reference Config File #
##########################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka.http {

  server {
    # The default value of the `Server` header to produce if no
```

```
    # explicit `Server`-header was included in a response.
    # If this value is the empty string and no header was included in
    # the request, no `Server` header will be rendered at all.
    server-header = akka-http/${akka.version}

    # The time after which an idle connection will be automatically closed.
    # Set to `infinite` to completely disable idle connection timeouts.
    idle-timeout = 60 s

    # The time period within which the TCP binding process must be completed.
    # Set to `infinite` to disable.
    bind-timeout = 1s

    # The maximum number of concurrently accepted connections when using the
    # `Http().bindAndHandle` methods.
    #
    # This setting doesn't apply to the `Http().bind` method which will still
    # deliver an unlimited backpressured stream of incoming connections.
    max-connections = 1024

    # The maximum number of requests that are accepted (and dispatched to
    # the application) on one single connection before the first request
    # has to be completed.
    # Incoming requests that would cause the pipelining limit to be exceeded
    # are not read from the connections socket so as to build up "back-pressure"
    # to the client via TCP flow control.
    # A setting of 1 disables HTTP pipelining, since only one request per
    # connection can be "open" (i.e. being processed by the application) at any
    # time. Set to higher values to enable HTTP pipelining.
    # This value must be > 0 and <= 1024.
    pipelining-limit = 16

    # Enables/disables the addition of a `Remote-Address` header
    # holding the clients (remote) IP address.
    remote-address-header = off

    # Enables/disables the addition of a `Raw-Request-URI` header holding the
    # original raw request URI as the client has sent it.
    raw-request-uri-header = off

    # Enables/disables automatic handling of HEAD requests.
    # If this setting is enabled the server dispatches HEAD requests as GET
    # requests to the application and automatically strips off all message
    # bodies from outgoing responses.
    # Note that, even when this setting is off the server will never send
    # out message bodies on responses to HEAD requests.
    transparent-head-requests = on

    # Enables/disables the returning of more detailed error messages to
    # the client in the error response.
    # Should be disabled for browser-facing APIs due to the risk of XSS attacks
    # and (probably) enabled for internal or non-browser APIs.
    # Note that akka-http will always produce log messages containing the full
    # error details.
    verbose-error-messages = off

    # The initial size of the buffer to render the response headers in.
    # Can be used for fine-tuning response rendering performance but probably
    # doesn't have to be fiddled with in most applications.
    response-header-size-hint = 512

    # The requested maximum length of the queue of incoming connections.
    # If the server is busy and the backlog is full the OS will start dropping
```

```
    # SYN-packets and connection attempts may fail. Note, that the backlog
    # size is usually only a maximum size hint for the OS and the OS can
    # restrict the number further based on global limits.
    backlog = 100

    # If this setting is empty the server only accepts requests that carry a
    # non-empty `Host` header. Otherwise it responds with `400 Bad Request`.
    # Set to a non-empty value to be used in lieu of a missing or empty `Host`
    # header to make the server accept such requests.
    # Note that the server will never accept HTTP/1.1 request without a `Host`
    # header, i.e. this setting only affects HTTP/1.1 requests with an empty
    # `Host` header as well as HTTP/1.0 requests.
    # Examples: `www.spray.io` or `example.com:8080`
    default-host-header = ""

    # Socket options to set for the listening socket. If a setting is left
    # undefined, it will use whatever the default on the system is.
    socket-options {
      so-receive-buffer-size = undefined
      so-send-buffer-size = undefined
      so-reuse-address = undefined
      so-traffic-class = undefined
      tcp-keep-alive = undefined
      tcp-oob-inline = undefined
      tcp-no-delay = undefined
    }

    # Modify to tweak parsing settings on the server-side only.
    parsing {
      # no overrides by default, see `akka.http.parsing` for default values
    }
  }

  client {
    # The default value of the `User-Agent` header to produce if no
    # explicit `User-Agent`-header was included in a request.
    # If this value is the empty string and no header was included in
    # the request, no `User-Agent` header will be rendered at all.
    user-agent-header = akka-http/${akka.version}

    # The time period within which the TCP connecting process must be completed.
    connecting-timeout = 10s

    # The time after which an idle connection will be automatically closed.
    # Set to `infinite` to completely disable idle timeouts.
    idle-timeout = 60 s

    # The initial size of the buffer to render the request headers in.
    # Can be used for fine-tuning request rendering performance but probably
    # doesn't have to be fiddled with in most applications.
    request-header-size-hint = 512

    # The proxy configurations to be used for requests with the specified
    # scheme.
    proxy {
      # Proxy settings for unencrypted HTTP requests
      # Set to 'none' to always connect directly, 'default' to use the system
      # settings as described in http://docs.oracle.com/javase/6/docs/technotes/guides/net/proxies
      # or specify the proxy host, port and non proxy hosts as demonstrated
      # in the following example:
      # http {
      #   host = myproxy.com
      #   port = 8080
```

```
    #   non-proxy-hosts = ["*.direct-access.net"]
    # }
    http = default

    # Proxy settings for HTTPS requests (currently unsupported)
    https = default
  }

  # Socket options to set for the listening socket. If a setting is left
  # undefined, it will use whatever the default on the system is.
  socket-options {
    so-receive-buffer-size = undefined
    so-send-buffer-size = undefined
    so-reuse-address = undefined
    so-traffic-class = undefined
    tcp-keep-alive = undefined
    tcp-oob-inline = undefined
    tcp-no-delay = undefined
  }

  # Modify to tweak parsing settings on the client-side only.
  parsing {
    # no overrides by default, see `akka.http.parsing` for default values
  }
}

host-connection-pool {
  # The maximum number of parallel connections that a connection pool to a
  # single host endpoint is allowed to establish. Must be greater than zero.
  max-connections = 4

  # The maximum number of times failed requests are attempted again,
  # (if the request can be safely retried) before giving up and returning an error.
  # Set to zero to completely disable request retries.
  max-retries = 5

  # The maximum number of open requests accepted into the pool across all
  # materializations of any of its client flows.
  # Protects against (accidentally) overloading a single pool with too many client flow material
  # Note that with N concurrent materializations the max number of open request in the pool
  # will never exceed N * max-connections * pipelining-limit.
  # Must be a power of 2 and > 0!
  max-open-requests = 32

  # The maximum number of requests that are dispatched to the target host in
  # batch-mode across a single connection (HTTP pipelining).
  # A setting of 1 disables HTTP pipelining, since only one request per
  # connection can be "in flight" at any time.
  # Set to higher values to enable HTTP pipelining.
  # This value must be > 0.
  # (Note that, independently of this setting, pipelining will never be done
  # on a connection that still has a non-idempotent request in flight.
  # See http://tools.ietf.org/html/rfc7230#section-6.3.2 for more info.)
  pipelining-limit = 1

  # The time after which an idle connection pool (without pending requests)
  # will automatically terminate itself. Set to `infinite` to completely disable idle timeouts.
  idle-timeout = 30 s

  # Modify to tweak client settings for host connection pools only.
  #
  # IMPORTANT:
  # Please note that this section mirrors `akka.http.client` however is used only for pool-base
```

```
    # such as `Http().superPool` or `Http().singleRequest`.
    client = {
      # The default value of the `User-Agent` header to produce if no
      # explicit `User-Agent`-header was included in a request.
      # If this value is the empty string and no header was included in
      # the request, no `User-Agent` header will be rendered at all.
      user-agent-header = akka-http/${akka.version}

      # The time period within which the TCP connecting process must be completed.
      connecting-timeout = 10s

      # The time after which an idle connection will be automatically closed.
      # Set to `infinite` to completely disable idle timeouts.
      idle-timeout = 60 s

      # The initial size of the buffer to render the request headers in.
      # Can be used for fine-tuning request rendering performance but probably
      # doesn't have to be fiddled with in most applications.
      request-header-size-hint = 512

      # The proxy configurations to be used for requests with the specified
      # scheme.
      proxy {
        # Proxy settings for unencrypted HTTP requests
        # Set to 'none' to always connect directly, 'default' to use the system
        # settings as described in http://docs.oracle.com/javase/6/docs/technotes/guides/net/proxi
        # or specify the proxy host, port and non proxy hosts as demonstrated
        # in the following example:
        # http {
        #   host = myproxy.com
        #   port = 8080
        #   non-proxy-hosts = ["*.direct-access.net"]
        # }
        http = default

        # Proxy settings for HTTPS requests (currently unsupported)
        https = default
      }

      # Socket options to set for the listening socket. If a setting is left
      # undefined, it will use whatever the default on the system is.
      socket-options {
        so-receive-buffer-size = undefined
        so-send-buffer-size = undefined
        so-reuse-address = undefined
        so-traffic-class = undefined
        tcp-keep-alive = undefined
        tcp-oob-inline = undefined
        tcp-no-delay = undefined
      }


      # IMPORTANT: Please note that this section is replicated in `client` and `server`.
      parsing {
        # no overrides by default, see `akka.http.parsing` for default values
      }
    }
  }

  # Modify to tweak default parsing settings.
  #
  # IMPORTANT:
  # Please note that this sections settings can be overriden by the corresponding settings in:
```

```
# `akka.http.server.parsing`, `akka.http.client.parsing` or `akka.http.http-connection-pool.clie
parsing {
  # The limits for the various parts of the HTTP message parser.
  max-uri-length             = 2k
  max-method-length          = 16
  max-response-reason-length = 64
  max-header-name-length     = 64
  max-header-value-length    = 8k
  max-header-count           = 64
  max-chunk-ext-length       = 256
  max-chunk-size             = 1m

  # Maximum content length which should not be exceeded by incoming HttpRequests.
  # For file uploads which use the entityBytes Source of an incoming HttpRequest it is safe to
  # set this to a very high value (or to `infinite` if feeling very adventurous) as the streamin
  # upload will be back-pressured properly by Akka Streams.
  # Please note however that this setting is a global property, and is applied to all incoming
  # not only file uploads consumed in a streaming fashion, so pick this limit wisely.
  max-content-length         = 8m

  # Sets the strictness mode for parsing request target URIs.
  # The following values are defined:
  #
  # `strict`: RFC3986-compliant URIs are required,
  #     a 400 response is triggered on violations
  #
  # `relaxed`: all visible 7-Bit ASCII chars are allowed
  #
  uri-parsing-mode = strict

  # Sets the parsing mode for parsing cookies.
  # The following value are defined:
  #
  # `rfc6265`: Only RFC6265-compliant cookies are parsed. Surrounding double-quotes are accepted
  #   automatically removed. Non-compliant cookies are silently discarded.
  # `raw`: Raw parsing allows any non-control character but ';' to appear in a cookie value. The
  #   post-processing applied, so that the resulting value string may contain any number of whit
  #   double quotes, or '=' characters at any position.
  #   The rules for parsing the cookie name are the same ones from RFC 6265.
  #
  cookie-parsing-mode = rfc6265

  # Enables/disables the logging of warning messages in case an incoming
  # message (request or response) contains an HTTP header which cannot be
  # parsed into its high-level model class due to incompatible syntax.
  # Note that, independently of this settings, akka-http will accept messages
  # with such headers as long as the message as a whole would still be legal
  # under the HTTP specification even without this header.
  # If a header cannot be parsed into a high-level model instance it will be
  # provided as a `RawHeader`.
  # If logging is enabled it is performed with the configured
  # `error-logging-verbosity`.
  illegal-header-warnings = on

  # Configures the verbosity with which message (request or response) parsing
  # errors are written to the application log.
  #
  # Supported settings:
  # `off`   : no log messages are produced
  # `simple`: a condensed single-line message is logged
  # `full`  : the full error details (potentially spanning several lines) are logged
  error-logging-verbosity = full
```

```
    # limits for the number of different values per header type that the
    # header cache will hold
    header-cache {
      default = 12
      Content-MD5 = 0
      Date = 0
      If-Match = 0
      If-Modified-Since = 0
      If-None-Match = 0
      If-Range = 0
      If-Unmodified-Since = 0
      User-Agent = 32
    }
  }
}
```

2.2.2 akka-http

```
########################################
# akka-http Reference Config File #
########################################

# This is the reference config file that contains all the default settings.
# Make your edits/overrides in your application.conf.

akka.http.routing {
  # Enables/disables the returning of more detailed error messages to the
  # client in the error response
  # Should be disabled for browser-facing APIs due to the risk of XSS attacks
  # and (probably) enabled for internal or non-browser APIs
  # (Note that akka-http will always produce log messages containing the full error details)
  verbose-error-messages = off

  # Enables/disables ETag and `If-Modified-Since` support for FileAndResourceDirectives
  file-get-conditional = on

  # Enables/disables the rendering of the "rendered by" footer in directory listings
  render-vanity-footer = yes

  # The maximum size between two requested ranges. Ranges with less space in between will be coale
  #
  # When multiple ranges are requested, a server may coalesce any of the ranges that overlap or th
  # by a gap that is smaller than the overhead of sending multiple parts, regardless of the order
  # corresponding byte-range-spec appeared in the received Range header field. Since the typical o
  # parts of a multipart/byteranges payload is around 80 bytes, depending on the selected represen
  # media type and the chosen boundary parameter length, it can be less efficient to transfer many
  # disjoint parts than it is to transfer the entire selected representation.
  range-coalescing-threshold = 80

  # The maximum number of allowed ranges per request.
  # Requests with more ranges will be rejected due to DOS suspicion.
  range-count-limit = 16

  # The maximum number of bytes per ByteString a decoding directive will produce
  # for an entity data stream.
  decode-max-bytes-per-chunk = 1m

  # Fully qualified config path which holds the dispatcher configuration
  # to be used by FlowMaterialiser when creating Actors for IO operations.
  file-io-dispatcher = ${akka.stream.blocking-io-dispatcher}
}
```

The other Akka HTTP modules do not offer any configuration via Typesafe Config.

2.3 Common Abstractions (Client- and Server-Side)

HTTP and related specifications define a great number of concepts and functionality that is not specific to either HTTP's client- or server-side since they are meaningful on both end of an HTTP connection. The documentation for their counterparts in Akka HTTP lives in this section rather than in the ones for the *Client-Side API*, *Low-Level Server-Side API* or *High-level Server-Side API*, which are specific to one side only.

2.3.1 HTTP Model

Akka HTTP model contains a deeply structured, fully immutable, case-class based model of all the major HTTP data structures, like HTTP requests, responses and common headers. It lives in the *akka-http-core* module and forms the basis for most of Akka HTTP's APIs.

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 all of the most relevant types in scope, 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

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

An `HttpRequest` consists of

- a method (GET, POST, etc.)
- a URI
- a seq of headers
- an entity (body data)
- a protocol

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

```
import HttpMethods._

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

// construct simple GET request to "/index" (implicit string to Uri conversion)
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._
import HttpCharsets._
val userData = ByteString("abc")
val authorization = headers.Authorization(BasicHttpCredentials("user", "pass"))
HttpRequest(
  PUT,
  uri = "/user",
  entity = HttpEntity(`text/plain` withCharset `UTF-8`, userData),
  headers = List(authorization),
  protocol = `HTTP/1.0`)
```

All parameters of `HttpRequest.apply` have default values set, so `headers` for example 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.

HttpResponse

An `HttpResponse` consists of

- a status code
- a seq of headers
- an entity (body data)
- a protocol

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))
```

In addition to the simple `HttpEntity` constructors which create an entity from a fixed `String` or `ByteString` as shown here the Akka HTTP model defines a number of subclasses of `HttpEntity` which allow body data to be specified as a stream of bytes.

HttpEntity

An `HttpEntity` carries the data bytes of a message together with its Content-Type and, if known, its Content-Length. In Akka HTTP there are five different kinds of entities which model the various ways that message content can be received or sent:

HttpEntity.Strict The simplest entity, which is used when all the entity are already available in memory. It wraps a plain `ByteString` and represents a standard, unchunked entity with a known `Content-Length`.

HttpEntity.Default The general, unchunked HTTP/1.1 message entity. It has a known length and presents its data as a `Source[ByteString]` which can be only materialized once. It is an error if the provided source doesn't produce exactly as many bytes as specified. The distinction of `Strict` and `Default` is an API-only one. One the wire, both kinds of entities look the same.

HttpEntity.Chunked The model for HTTP/1.1 chunked content (i.e. sent with `Transfer-Encoding: chunked`). The content length is unknown and the individual chunks are presented as a `Source[HttpEntity.ChunkStreamPart]`. A `ChunkStreamPart` is either a non-empty `Chunk` or a `LastChunk` containing optional trailer headers. The stream consists of zero or more `Chunked` parts and can be terminated by an optional `LastChunk` part.

HttpEntity.CloseDelimited An unchunked entity of unknown length that is implicitly delimited by closing the connection (`Connection: close`). The content data are presented as a `Source[ByteString]`. Since the connection must be closed after sending an entity of this type it can only be used on the server-side for sending a response. Also, the main purpose of `CloseDelimited` entities is compatibility with HTTP/1.0 peers, which do not support chunked transfer encoding. If you are building a new application and are not constrained by legacy requirements you shouldn't rely on `CloseDelimited` entities, since implicit terminate-by-connection-close is not a robust way of signaling response end, especially in the presence of proxies. Additionally this type of entity prevents connection reuse which can seriously degrade performance. Use `HttpEntity.Chunked` instead!

HttpEntity.IndefiniteLength A streaming entity of unspecified length for use in a `Multipart.BodyPart`.

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, the general method `HttpEntity.dataBytes` is provided which returns a `Source[ByteString, Any]` that 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 already available in memory (e.g. as a `String` or `ByteString`)
- Use `Default` if the data is generated by a streaming data source and the size of the data is known
- Use `Chunked` for an entity of unknown length
- Use `CloseDelimited` for a response as a legacy alternative to `Chunked` if the client doesn't support chunked transfer encoding. Otherwise use `Chunked`!
- In a `Multipart.Bodypart` use `IndefiniteLength` for content of unknown length.

Caution: When you receive a non-strict message from a connection then additional data are only read from the network when you request them by consuming the entity data stream. This means that, if you *don't* consume the entity stream then the connection will effectively be stalled. In particular no subsequent message (request or response) will be read from the connection as the entity of the current message "blocks" the stream. Therefore you must make sure that you always consume the entity data, even in the case that you are not actually interested in it!

Limiting message entity length

All message entities that Akka HTTP reads from the network automatically get a length verification check attached to them. This check makes sure that the total entity size is less than or equal to the configured `max-content-length` [1], which is an important defense against certain Denial-of-Service attacks. However, a single global limit for all requests (or responses) is often too inflexible for applications that need to allow large limits for *some* requests (or responses) but want to clamp down on all messages not belonging into that group.

In order to give you maximum flexibility in defining entity size limits according to your needs the `HttpEntity` features a `withSizeLimit` method, which lets you adjust the globally configured maximum size for this particular entity, be it to increase or decrease any previously set value. This means that your application will receive all requests (or responses) from the HTTP layer, even the ones whose `Content-Length` exceeds the configured limit (because you might want to increase the limit yourself). Only when the actual data stream `Source` contained in the entity is materialized will the boundary checks be actually applied. In case the length verification fails the respective stream will be terminated with an `EntityStreamSizeException` either directly at materialization time (if the `Content-Length` is known) or whenever more data bytes than allowed have been read.

When called on `Strict` entities the `withSizeLimit` method will return the entity itself if the length is within the bound, otherwise a `Default` entity with a single element data stream. This allows for potential refinement of the entity size limit at a later point (before materialization of the data stream).

By default all message entities produced by the HTTP layer automatically carry the limit that is defined in the application's `max-content-length` config setting. If the entity is transformed in a way that changes the content-length and then another limit is applied then this new limit will be evaluated against the new content-length. If the entity is transformed in a way that changes the content-length and no new limit is applied then the previous limit will be applied against the previous content-length. Generally this behavior should be in line with your expectations.

Special processing for HEAD requests

RFC 7230 defines very clear rules for the entity length of HTTP messages.

Especially this rule requires special treatment in Akka HTTP:

> Any response to a HEAD request and any response with a 1xx (Informational), 204 (No Content), or 304 (Not Modified) status code is always terminated by the first empty line after the header fields, regardless of the header fields present in the message, and thus cannot contain a message body.

Responses to HEAD requests introduce the complexity that *Content-Length* or *Transfer-Encoding* headers can be present but the entity is empty. This is modeled by allowing *HttpEntity.Default* and *HttpEntity.Chunked* to be used for HEAD responses with an empty data stream.

Also, when a HEAD response has an *HttpEntity.CloseDelimited* entity the Akka HTTP implementation will *not* close the connection after the response has been sent. This allows the sending of HEAD responses without *Content-Length* header across persistent HTTP connections.

[1] *akka.http.parsing.max-content-length* (applying to server- as well as client-side), *akka.http.server.parsing.max-content-length* (server-side only), *akka.http.client.parsing.max-content-length* (client-side only) or *akka.http.host-connection-pool.client.parsing.max-content-length* (only host-connection-pools)

Header Model

Akka HTTP contains a rich model of the most 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` (which is essentially a String/String name/value pair).

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"))

// custom type
case class User(name: String, pass: String)

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

HTTP Headers

When the Akka HTTP server receives an HTTP request it tries to parse all its headers into their respective model classes. Independently of whether this succeeds or not, the HTTP layer 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 headers have special status in HTTP and are therefore treated differently from "regular" headers:

Content-Type The Content-Type of an HTTP message is modeled as the `contentType` field of the `HttpEntity`. The `Content-Type` header therefore doesn't appear in the `headers` sequence of a message. Also, a `Content-Type` header instance that is explicitly added to the `headers` of a request or response will not be rendered onto the wire and trigger a warning being logged instead!

Transfer-Encoding Messages with `Transfer-Encoding: chunked` are represented via the `HttpEntity.Chunked` entity. As such chunked messages that do not have another deeper nested transfer encoding will not have a `Transfer-Encoding` header in their `headers` sequence. Similarly, a `Transfer-Encoding` header instance that is explicitly added to the `headers` of a request or response will not be rendered onto the wire and trigger a warning being logged instead!

Content-Length The content length of a message is modelled via its *HttpEntity*. As such no `Content-Length` header will ever be part of a message's `header` sequence. Similarly, a `Content-Length` header instance that is explicitly added to the `headers` of a request or response will not be rendered onto the wire and trigger a warning being logged instead!

Server A `Server` header is usually added automatically to any response and its value can be configured via the `akka.http.server.server-header` setting. Additionally an application can override the configured header with a custom one by adding it to the response's `header` sequence.

User-Agent A `User-Agent` header is usually added automatically to any request and its value can be configured via the `akka.http.client.user-agent-header` setting. Additionally an application can override the configured header with a custom one by adding it to the request's `header` sequence.

Date The `Date` response header is added automatically but can be overridden by supplying it manually.

Connection On the server-side Akka HTTP watches for explicitly added `Connection: close` response headers and as such honors the potential wish of the application to close the connection after the respective

response has been sent out. The actual logic for determining whether to close the connection is quite involved. It takes into account the request's method, protocol and potential `Connection` header as well as the response's protocol, entity and potential `Connection` header. See this test for a full table of what happens when.

Custom Headers

Sometimes you may need to model a custom header type which is not part of HTTP and still be able to use it as convienient as is possible with the built-in types.

Because of the number of ways one may interact with headers (i.e. try to match a `CustomHeader` against a `RawHeader` or the other way around etc), a helper trait for custom Header types and their companions classes are provided by Akka HTTP. Thanks to extending `ModeledCustomHeader` instead of the plain `CustomHeader` such header can be matched

```
object ApiTokenHeader extends ModeledCustomHeaderCompanion[ApiTokenHeader] {
  override val name = "apiKey"
  override def parse(value: String) = Try(new ApiTokenHeader(value))
}
final class ApiTokenHeader(token: String) extends ModeledCustomHeader[ApiTokenHeader] {
  override val companion = ApiTokenHeader
  override def value: String = token
}
```

Which allows the this CustomHeader to be used in the following scenarios:

```
val ApiTokenHeader(t1) = ApiTokenHeader("token")
t1 should ===("token")

val RawHeader(k2, v2) = ApiTokenHeader("token")
k2 should ===("apiKey")
v2 should ===("token")

// will match, header keys are case insensitive
val ApiTokenHeader(v3) = RawHeader("APIKEY", "token")
v3 should ===("token")

intercept[MatchError] {
  // won't match, different header name
  val ApiTokenHeader(v4) = DifferentHeader("token")
}

intercept[MatchError] {
  // won't match, different header name
  val RawHeader("something", v5) = DifferentHeader("token")
}

intercept[MatchError] {
  // won't match, different header name
  val ApiTokenHeader(v6) = RawHeader("different", "token")
}
```

Including usage within the header directives like in the following *headerValuePF* example:

```
def extractFromCustomHeader = headerValuePF {
  case t @ ApiTokenHeader(token) ⇒ s"extracted> $t"
  case raw: RawHeader            ⇒ s"raw> $raw"
}

val routes = extractFromCustomHeader { s ⇒
  complete(s)
}
```

```
Get().withHeaders(RawHeader("apiKey", "TheKey")) ~> routes ~> check {
  status should ===(StatusCodes.OK)
  responseAs[String] should ===("extracted> apiKey: TheKey")
}

Get().withHeaders(RawHeader("somethingElse", "TheKey")) ~> routes ~> check {
  status should ===(StatusCodes.OK)
  responseAs[String] should ===("raw> somethingElse: TheKey")
}

Get().withHeaders(ApiTokenHeader("TheKey")) ~> routes ~> check {
  status should ===(StatusCodes.OK)
  responseAs[String] should ===("extracted> apiKey: TheKey")
}
```

One can also directly extend `CustomHeader` which requires less boilerplate, however that has the downside of matching against `RawHeader` instances not working out-of-the-box, thus limiting its usefulnes in the routing layer of Akka HTTP. For only rendering such header however it would be enough.

Note: When defining custom headers, prefer to extend `ModeledCustomHeader` instead of `CustomHeader` directly as it will automatically make your header abide all the expected pattern matching semantics one is accustomed to when using built-in types (such as matching a custom header against a `RawHeader` as is often the case in routing layers of Akka HTTP applications).

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.

Note: Various parsing and rendering settings are available to tweak in the configuration under `akka.http.client[.parsing]`, `akka.http.server[.parsing]` and `akka.http.host-connection-pool[.client.parsing]`, with defaults for all of these being defined in the `akka.http.parsing` configuration section.

For example, if you want to change a parsing setting for all components, you can set the `akka.http.parsing.illegal-header-warnings = off` value. However this setting can be stil overriden by the more specific sections, like for example `akka.http.server.parsing.illegal-header-warnings = on`. In this case both `client` and `host-connection-pool` APIs will see the setting `off`, however the server will see `on`.

In the case of `akka.http.host-connection-pool.client` settings, they default to settings set in `akka.http.client`, and can override them if needed. This is useful, since both `client` and `host-connection-pool` APIs, such as the Client API `Http().outgoingConnection` or the Host Connection Pool APIs `Http().singleRequest` or `Http().superPool`, usually need the same settings, however the `server` most likely has a very different set of settings.

2.3.2 Marshalling

"Marshalling" is the process of converting a higher-level (object) structure into some kind of lower-level representation, often a "wire format". Other popular names for it are "Serialization" or "Pickling".

In Akka HTTP "Marshalling" means the conversion of an object of type `T` into a lower-level target type, e.g. a `MessageEntity` (which forms the "entity body" of an HTTP request or response) or a full `HttpRequest` or `HttpResponse`.

Basic Design

Marshalling of instances of type `A` into instances of type `B` is performed by a `Marshaller[A, B]`. Akka HTTP also predefines a number of helpful aliases for the types of marshallers that you'll likely work with most:

```
type ToEntityMarshaller[T] = Marshaller[T, MessageEntity]
type ToHeadersAndEntityMarshaller[T] = Marshaller[T, (immutable.Seq[HttpHeader], MessageEntity)]
type ToResponseMarshaller[T] = Marshaller[T, HttpResponse]
type ToRequestMarshaller[T] = Marshaller[T, HttpRequest]
```

Contrary to what you might initially expect `Marshaller[A, B]` is not a plain function `A => B` but rather essentially a function `A => Future[List[Marshalling[B]]]`. Let's dissect this rather complicated looking signature piece by piece to understand what marshallers are designed this way. Given an instance of type `A` a `Marshaller[A, B]` produces:

1. A `Future`: This is probably quite clear. Marshallers are not required to synchronously produce a result, so instead they return a future, which allows for asynchronicity in the marshalling process.

2. of `List`: Rather than only a single target representation for `A` marshallers can offer several ones. Which one will be rendered onto the wire in the end is decided by content negotiation. For example, the `ToEntityMarshaller[OrderConfirmation]` might offer a JSON as well as an XML representation. The client can decide through the addition of an `Accept` request header which one is preferred. If the client doesn't express a preference the first representation is picked.

3. of `Marshalling[B]`: Rather than returning an instance of `B` directly marshallers first produce a `Marshalling[B]`. This allows for querying the `MediaType` and potentially the `HttpCharset` that the marshaller will produce before the actual marshalling is triggered. Apart from enabling content negotiation this design allows for delaying the actual construction of the marshalling target instance to the very last moment when it is really needed.

This is how `Marshalling` is defined:

```
/**
 * Describes one possible option for marshalling a given value.
 */
sealed trait Marshalling[+A] {
  def map[B](f: A ⇒ B): Marshalling[B]
}

object Marshalling {

  /**
   * A Marshalling to a specific [[ContentType]].
   */
  final case class WithFixedContentType[A](contentType: ContentType,
                                           marshal: () ⇒ A) extends Marshalling[A] {
    def map[B](f: A ⇒ B): WithFixedContentType[B] = copy(marshal = () ⇒ f(marshal()))
  }

  /**
   * A Marshalling to a specific [[MediaType]] with a flexible charset.
   */
  final case class WithOpenCharset[A](mediaType: MediaType.WithOpenCharset,
                                      marshal: HttpCharset ⇒ A) extends Marshalling[A] {
    def map[B](f: A ⇒ B): WithOpenCharset[B] = copy(marshal = cs ⇒ f(marshal(cs)))
  }

  /**
   * A Marshalling to an unknown MediaType and charset.
   * Circumvents content negotiation.
   */
  final case class Opaque[A](marshal: () ⇒ A) extends Marshalling[A] {
    def map[B](f: A ⇒ B): Opaque[B] = copy(marshal = () ⇒ f(marshal()))
```

```
  }
}
```

Predefined Marshallers

Akka HTTP already predefines a number of marshallers for the most common types. Specifically these are:

- PredefinedToEntityMarshallers
  - `Array[Byte]`
  - `ByteString`
  - `Array[Char]`
  - `String`
  - `akka.http.scaladsl.model.FormData`
  - `akka.http.scaladsl.model.MessageEntity`
  - `T <: akka.http.scaladsl.model.Multipart`
- PredefinedToResponseMarshallers
  - `T`, if a `ToEntityMarshaller[T]` is available
  - `HttpResponse`
  - `StatusCode`
  - `(StatusCode, T)`, if a `ToEntityMarshaller[T]` is available
  - `(Int, T)`, if a `ToEntityMarshaller[T]` is available
  - `(StatusCode, immutable.Seq[HttpHeader], T)`, if a `ToEntityMarshaller[T]` is available
  - `(Int, immutable.Seq[HttpHeader], T)`, if a `ToEntityMarshaller[T]` is available
- PredefinedToRequestMarshallers
  - `HttpRequest`
  - `Uri`
  - `(HttpMethod, Uri, T)`, if a `ToEntityMarshaller[T]` is available
  - `(HttpMethod, Uri, immutable.Seq[HttpHeader], T)`, if a `ToEntityMarshaller[T]` is available
- GenericMarshallers
  - `Marshaller[Throwable, T]`
  - `Marshaller[Option[A], B]`, if a `Marshaller[A, B]` and an `EmptyValue[B]` is available
  - `Marshaller[Either[A1, A2], B]`, if a `Marshaller[A1, B]` and a `Marshaller[A2, B]` is available
  - `Marshaller[Future[A], B]`, if a `Marshaller[A, B]` is available
  - `Marshaller[Try[A], B]`, if a `Marshaller[A, B]` is available

Implicit Resolution

The marshalling infrastructure of Akka HTTP relies on a type-class based approach, which means that `Marshaller` instances from a certain type `A` to a certain type `B` have to be available implicitly.

The implicits for most of the predefined marshallers in Akka HTTP are provided through the companion object of the `Marshaller` trait. This means that they are always available and never need to be explicitly imported. Additionally, you can simply "override" them by bringing your own custom version into local scope.

Custom Marshallers

Akka HTTP gives you a few convenience tools for constructing marshallers for your own types. Before you do that you need to think about what kind of marshaller you want to create. If all your marshaller needs to produce is a `MessageEntity` then you should probably provide a `ToEntityMarshaller[T]`. The advantage here is that it will work on both the client- as well as the server-side since a `ToResponseMarshaller[T]` as well as a `ToRequestMarshaller[T]` can automatically be created if a `ToEntityMarshaller[T]` is available.

If, however, your marshaller also needs to set things like the response status code, the request method, the request URI or any headers then a `ToEntityMarshaller[T]` won't work. You'll need to fall down to providing a `ToResponseMarshaller[T]` or a `ToRequestMarshaller[T]` directly.

For writing you own marshallers you won't have to "manually" implement the `Marshaller` trait directly. Rather, it should be possible to use one of the convenience construction helpers defined on the `Marshaller` companion:

```
object Marshaller
  extends GenericMarshallers
  with PredefinedToEntityMarshallers
  with PredefinedToResponseMarshallers
  with PredefinedToRequestMarshallers {

  /**
   * Creates a [[Marshaller]] from the given function.
   */
  def apply[A, B](f: ExecutionContext ⇒ A ⇒ Future[List[Marshalling[B]]]): Marshaller[A, B] =
    new Marshaller[A, B] {
      def apply(value: A)(implicit ec: ExecutionContext) =
        try f(ec)(value)
        catch { case NonFatal(e) ⇒ FastFuture.failed(e) }
    }

  /**
   * Helper for creating a [[Marshaller]] using the given function.
   */
  def strict[A, B](f: A ⇒ Marshalling[B]): Marshaller[A, B] =
    Marshaller { _ ⇒ a ⇒ FastFuture.successful(f(a) :: Nil) }

  /**
   * Helper for creating a "super-marshaller" from a number of "sub-marshallers".
   * Content-negotiation determines, which "sub-marshaller" eventually gets to do the job.
   */
  def oneOf[A, B](marshallers: Marshaller[A, B]*): Marshaller[A, B] =
    Marshaller { implicit ec ⇒ a ⇒ FastFuture.sequence(marshallers.map(_(a))).fast.map(_.flatten

  /**
   * Helper for creating a "super-marshaller" from a number of values and a function producing "su
   * from these values. Content-negotiation determines, which "sub-marshaller" eventually gets to
   */
  def oneOf[T, A, B](values: T*)(f: T ⇒ Marshaller[A, B]): Marshaller[A, B] =
    oneOf(values map f: _*)

  /**
   * Helper for creating a synchronous [[Marshaller]] to content with a fixed charset from the gi
```

```
   */
  def withFixedContentType[A, B](contentType: ContentType)(marshal: A ⇒ B): Marshaller[A, B] =
    strict { value ⇒ Marshalling.WithFixedContentType(contentType, () ⇒ marshal(value)) }

  /**
   * Helper for creating a synchronous [[Marshaller]] to content with a negotiable charset from th
   */
  def withOpenCharset[A, B](mediaType: MediaType.WithOpenCharset)(marshal: (A, HttpCharset) ⇒ B):
    strict { value ⇒ Marshalling.WithOpenCharset(mediaType, charset ⇒ marshal(value, charset)) }

  /**
   * Helper for creating a synchronous [[Marshaller]] to non-negotiable content from the given fu
   */
  def opaque[A, B](marshal: A ⇒ B): Marshaller[A, B] =
    strict { value ⇒ Marshalling.Opaque(() ⇒ marshal(value)) }

  /**
   * Helper for creating a [[Marshaller]] combined of the provided `marshal` function
   * and an implicit Marshaller which is able to produce the required final type.
   */
  def combined[A, B, C](marshal: A ⇒ B)(implicit m2: Marshaller[B, C]): Marshaller[A, C] =
    Marshaller[A, C] { ec ⇒ a ⇒ m2.compose(marshal).apply(a)(ec) }
}
```

Deriving Marshallers

Sometimes you can save yourself some work by reusing existing marshallers for your custom ones. The idea is to "wrap" an existing marshaller with some logic to "re-target" it to your type.

In this regard wrapping a marshaller can mean one or both of the following two things:

- Transform the input before it reaches the wrapped marshaller
- Transform the output of the wrapped marshaller

For the latter (transforming the output) you can use `baseMarshaller.map`, which works exactly as it does for functions. For the former (transforming the input) you have four alternatives:

- `baseMarshaller.compose`
- `baseMarshaller.composeWithEC`
- `baseMarshaller.wrap`
- `baseMarshaller.wrapWithEC`

`compose` works just like it does for functions. `wrap` is a compose that allows you to also change the `ContentType` that the marshaller marshals to. The `...WithEC` variants allow you to receive an `ExecutionContext` internally if you need one, without having to depend on one being available implicitly at the usage site.

Using Marshallers

In many places throughput Akka HTTP marshallers are used implicitly, e.g. when you define how to *complete* a request using the *Routing DSL*.

However, you can also use the marshalling infrastructure directly if you wish, which can be useful for example in tests. The best entry point for this is the `akka.http.scaladsl.marshalling.Marshal` object, which you can use like this:

```
import scala.concurrent.Await
import scala.concurrent.duration._
import akka.http.scaladsl.marshalling.Marshal
```

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

import system.dispatcher // ExecutionContext

val string = "Yeah"
val entityFuture = Marshal(string).to[MessageEntity]
val entity = Await.result(entityFuture, 1.second) // don't block in non-test code!
entity.contentType shouldEqual ContentTypes.`text/plain(UTF-8)`

val errorMsg = "Easy, pal!"
val responseFuture = Marshal(420 -> errorMsg).to[HttpResponse]
val response = Await.result(responseFuture, 1.second) // don't block in non-test code!
response.status shouldEqual StatusCodes.EnhanceYourCalm
response.entity.contentType shouldEqual ContentTypes.`text/plain(UTF-8)`

val request = HttpRequest(headers = List(headers.Accept(MediaTypes.`application/json`)))
val responseText = "Plaintext"
val respFuture = Marshal(responseText).toResponseFor(request) // with content negotiation!
a[Marshal.UnacceptableResponseContentTypeException] should be thrownBy {
  Await.result(respFuture, 1.second) // client requested JSON, we only have text/plain!
}
```

2.3.3 Unmarshalling

"Unmarshalling" is the process of converting some kind of a lower-level representation, often a "wire format", into a higher-level (object) structure. Other popular names for it are "Deserialization" or "Unpickling".

In Akka HTTP "Unmarshalling" means the conversion of a lower-level source object, e.g. a `MessageEntity` (which forms the "entity body" of an HTTP request or response) or a full `HttpRequest` or `HttpResponse`, into an instance of type `T`.

Basic Design

Unmarshalling of instances of type `A` into instances of type `B` is performed by an `Unmarshaller[A, B]`. Akka HTTP also predefines a number of helpful aliases for the types of unmarshallers that you'll likely work with most:

```
type FromEntityUnmarshaller[T] = Unmarshaller[HttpEntity, T]
type FromMessageUnmarshaller[T] = Unmarshaller[HttpMessage, T]
type FromResponseUnmarshaller[T] = Unmarshaller[HttpResponse, T]
type FromRequestUnmarshaller[T] = Unmarshaller[HttpRequest, T]
type FromStringUnmarshaller[T] = Unmarshaller[String, T]
type FromStrictFormFieldUnmarshaller[T] = Unmarshaller[StrictForm.Field, T]
```

At its core an `Unmarshaller[A, B]` is very similar to a function `A => Future[B]` and as such quite a bit simpler than its *marshalling* counterpart. The process of unmarshalling does not have to support content negotiation which saves two additional layers of indirection that are required on the marshalling side.

Predefined Unmarshallers

Akka HTTP already predefines a number of marshallers for the most common types. Specifically these are:

- PredefinedFromStringUnmarshallers
  - `Byte`
  - `Short`
  - `Int`
  - `Long`

  - `Float`
  - `Double`
  - `Boolean`
- PredefinedFromEntityUnmarshallers
  - `Array[Byte]`
  - `ByteString`
  - `Array[Char]`
  - `String`
  - `akka.http.scaladsl.model.FormData`
- GenericUnmarshallers
  - `Unmarshaller[T, T]` (identity unmarshaller)
  - `Unmarshaller[Option[A], B]`, if an `Unmarshaller[A, B]` is available
  - `Unmarshaller[A, Option[B]]`, if an `Unmarshaller[A, B]` is available

Implicit Resolution

The unmarshalling infrastructure of Akka HTTP relies on a type-class based approach, which means that `Unmarshaller` instances from a certain type `A` to a certain type `B` have to be available implicitly.

The implicits for most of the predefined unmarshallers in Akka HTTP are provided through the companion object of the `Unmarshaller` trait. This means that they are always available and never need to be explicitly imported. Additionally, you can simply "override" them by bringing your own custom version into local scope.

Custom Unmarshallers

Akka HTTP gives you a few convenience tools for constructing unmarshallers for your own types. Usually you won't have to "manually" implement the `Unmarshaller` trait directly. Rather, it should be possible to use one of the convenience construction helpers defined on the `Marshaller` companion:

```
/**
 * Creates an `Unmarshaller` from the given function.
 */
def apply[A, B](f: ExecutionContext ⇒ A ⇒ Future[B]): Unmarshaller[A, B] =
  withMaterializer(ec => _ => f(ec))

def withMaterializer[A, B](f: ExecutionContext ⇒ Materializer => A ⇒ Future[B]): Unmarshaller[A,
  new Unmarshaller[A, B] {
    def apply(a: A)(implicit ec: ExecutionContext, materializer: Materializer) =
      try f(ec)(materializer)(a)
      catch { case NonFatal(e) ⇒ FastFuture.failed(e) }
  }

/**
 * Helper for creating a synchronous `Unmarshaller` from the given function.
 */
def strict[A, B](f: A ⇒ B): Unmarshaller[A, B] = Unmarshaller(_ => a ⇒ FastFuture.successful(f(a

/**
 * Helper for creating a "super-unmarshaller" from a sequence of "sub-unmarshallers", which are t
 * in the given order. The first successful unmarshalling of a "sub-unmarshallers" is the one proc
 * "super-unmarshaller".
 */
def firstOf[A, B](unmarshallers: Unmarshaller[A, B]*): Unmarshaller[A, B] = //...
```

Deriving Unmarshallers

Sometimes you can save yourself some work by reusing existing unmarshallers for your custom ones. The idea is to "wrap" an existing unmarshaller with some logic to "re-target" it to your type.

Usually what you want to do is to transform the output of some existing unmarshaller and convert it to your type. For this type of unmarshaller transformation Akka HTTP defines these methods:

- `baseUnmarshaller.transform`
- `baseUnmarshaller.map`
- `baseUnmarshaller.mapWithInput`
- `baseUnmarshaller.flatMap`
- `baseUnmarshaller.flatMapWithInput`
- `baseUnmarshaller.recover`
- `baseUnmarshaller.withDefaultValue`
- `baseUnmarshaller.mapWithCharset` (only available for FromEntityUnmarshallers)
- `baseUnmarshaller.forContentTypes` (only available for FromEntityUnmarshallers)

The method signatures should make their semantics relatively clear.

Using Unmarshallers

In many places throughput Akka HTTP unmarshallers are used implicitly, e.g. when you want to access the *entity* of a request using the *Routing DSL*.

However, you can also use the unmarshalling infrastructure directly if you wish, which can be useful for example in tests. The best entry point for this is the `akka.http.scaladsl.unmarshalling.Unmarshal` object, which you can use like this:

```
import akka.http.scaladsl.unmarshalling.Unmarshal
import system.dispatcher // ExecutionContext
implicit val materializer: Materializer = ActorMaterializer()

import scala.concurrent.Await
import scala.concurrent.duration._

val intFuture = Unmarshal("42").to[Int]
val int = Await.result(intFuture, 1.second) // don't block in non-test code!
int shouldEqual 42

val boolFuture = Unmarshal("off").to[Boolean]
val bool = Await.result(boolFuture, 1.second) // don't block in non-test code!
bool shouldBe false
```

2.3.4 Encoding / Decoding

The HTTP spec defines a `Content-Encoding` header, which signifies whether the entity body of an HTTP message is "encoded" and, if so, by which algorithm. The only commonly used content encodings are compression algorithms.

Currently Akka HTTP supports the compression and decompression of HTTP requests and responses with the `gzip` or `deflate` encodings. The core logic for this lives in the akka.http.scaladsl.coding package.

The support is not enabled automatically, but must be explicitly requested. For enabling message encoding/decoding with *Routing DSL* see the *CodingDirectives*.

2.3.5 JSON Support

Akka HTTP's *marshalling* and *unmarshalling* infrastructure makes it rather easy to seamlessly support specific wire representations of your data objects, like JSON, XML or even binary encodings.

For JSON Akka HTTP currently provides support for spray-json right out of the box through it's `akka-http-spray-json` module.

Other JSON libraries are supported by the community. See the list of current community extensions for Akka HTTP.

spray-json Support

The SprayJsonSupport trait provides a `FromEntityUnmarshaller[T]` and `ToEntityMarshaller[T]` for every type `T` that an implicit `spray.json.RootJsonReader` and/or `spray.json.RootJsonWriter` (respectively) is available for.

This is how you enable automatic support for (un)marshalling from and to JSON with spray-json:

1. Add a library dependency onto `"com.typesafe.akka" %% "akka-http-spray-json-experimental" % "2.0.2"`.
2. `import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._` or mix in the `akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport` trait.
3. Provide a `RootJsonFormat[T]` for your type and bring it into scope. Check out the spray-json documentation for more info on how to do this.

Once you have done this (un)marshalling between JSON and your type `T` should work nicely and transparently.

```
import spray.json._

// domain model
final case class Item(name: String, id: Long)
final case class Order(items: List[Item])

// collect your json format instances into a support trait:
trait JsonSupport extends SprayJsonSupport with DefaultJsonProtocol {
  implicit val itemFormat = jsonFormat2(Item)
  implicit val orderFormat = jsonFormat1(Order) // contains List[Item]
}

// use it wherever json (un)marshalling is needed
class MyJsonService extends Directives with JsonSupport {

  // format: OFF
  val route =
    get {
      pathSingleSlash {
        complete {
          Item("thing", 42) // will render as JSON
        }
      }
    } ~
    post {
      entity(as[Order]) { order => // will unmarshal JSON to Order
        val itemsCount = order.items.size
        val itemNames = order.items.map(_.name).mkString(", ")
        complete(s"Ordered $itemsCount items: $itemNames")
      }
    }
  // format: ON
}
```

```
  }
}
```

To learn more about how spray-json works please refer to its documentation.

2.3.6 XML Support

Akka HTTP's *marshalling* and *unmarshalling* infrastructure makes it rather easy to seamlessly support specific wire representations of your data objects, like JSON, XML or even binary encodings.

For XML Akka HTTP currently provides support for Scala XML right out of the box through it's `akka-http-xml` module.

Scala XML Support

The ScalaXmlSupport trait provides a `FromEntityUnmarshaller[NodeSeq]` and `ToEntityMarshaller[NodeSeq]` that you can use directly or build upon.

This is how you enable support for (un)marshalling from and to JSON with Scala XML `NodeSeq`:

1. Add a library dependency onto `"com.typesafe.akka" %% "akka-http-xml-experimental" % "1.x"`.
2. `import akka.http.scaladsl.marshallers.xml.ScalaXmlSupport._` or mix in the `akka.http.scaladsl.marshallers.xml.ScalaXmlSupport` trait.

Once you have done this (un)marshalling between XML and `NodeSeq` instances should work nicely and transparently.

2.4 Low-Level Server-Side API

Apart from the *HTTP Client* Akka HTTP also provides an embedded, Reactive-Streams-based, fully asynchronous HTTP/1.1 server implemented on top of *Akka Stream*.

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 API
- Optional SSL/TLS encryption
- Websocket support

The server-side components of Akka HTTP are split into two layers:

1. The basic low-level server implementation in the `akka-http-core` module
2. Higher-level functionality in the `akka-http` module

The low-level server (1) is scoped with a clear focus on the essential functionality of an HTTP/1.1 server:

- Connection management
- Parsing and rendering of 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 higher layers, they are not implemented by the `akka-http-core`-level server itself. Apart from general focus this design keeps the server core small and light-weight as well as easy to understand and maintain.

Depending on your needs you can either use the low-level API directly or rely on the high-level *Routing DSL* which can make the definition of more complex service logic much easier.

2.4.1 Streams and HTTP

The Akka HTTP server is implemented on top of *Akka Stream* and makes heavy use of it - in its implementation as well as on all levels of its API.

On the connection level Akka HTTP offers basically the same kind of interface as *Akka Stream IO*: A socket binding is represented as a stream of incoming connections. The application pulls connections from this stream source and, for each of them, provides a `Flow[HttpRequest, HttpResponse, _]` to "translate" requests into responses.

Apart from regarding a socket bound on the server-side as a `Source[IncomingConnection]` and each connection as a `Source[HttpRequest]` with a `Sink[HttpResponse]` the stream abstraction is also present inside a single HTTP message: The entities of HTTP requests and responses are generally modeled as a `Source[ByteString]`. See also the *HTTP Model* for more information on how HTTP messages are represented in Akka HTTP.

2.4.2 Starting and Stopping

On the most basic level an Akka HTTP server is bound by invoking the `bind` method of the akka.http.scaladsl.Http extension:

```
import akka.http.scaladsl.Http
import akka.stream.ActorMaterializer
import akka.stream.scaladsl._

implicit val system = ActorSystem()
implicit val materializer = ActorMaterializer()
implicit val ec = system.dispatcher

val serverSource: Source[Http.IncomingConnection, Future[Http.ServerBinding]] =
  Http().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 for the definition of 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[Http.IncomingConnection]` which must be drained by the application in order to accept incoming connections. The actual binding is not performed before this source is materialized as part of a processing pipeline. In case the bind fails (e.g. because the port is already busy) the materialized stream will immediately be terminated with a respective exception. The binding is released (i.e. the underlying socket unbound) when the subscriber of the incoming connection source has cancelled its subscription. Alternatively one can use the `unbind()` method of the `Http.ServerBinding` instance that is created as part of the connection source's materialization process. The `Http.ServerBinding` also provides a way to get a hold of the actual local address of the bound socket, which is useful for example when binding to port zero (and thus letting the OS pick an available port).

2.4.3 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 `handleWithXXX` methods with a handler, which can either be

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

Here is a complete example:

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.model.HttpMethods._
import akka.http.scaladsl.model._
import akka.stream.ActorMaterializer
import akka.stream.scaladsl.Sink

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

val serverSource = Http().bind(interface = "localhost", port = 8080)

val requestHandler: HttpRequest => HttpResponse = {
  case HttpRequest(GET, Uri.Path("/"), _, _, _) =>
    HttpResponse(entity = HttpEntity(ContentTypes.`text/html(UTF-8)`,
      "<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 example, 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 many other ways of providing a request handler 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). When relying on `handleWithSyncHandler` or `handleWithAsyncHandler`, or the `map` or `mapAsync` stream operators, this requirement will be automatically fulfilled.

Streaming Request/Response Entities

Streaming of HTTP message entities is supported through subclasses of `HttpEntity`. The application needs to be able to deal with streamed entities when receiving a request as well as, in many cases, when constructing responses. See *HttpEntity* for a description of the alternatives.

If you rely on the *Marshalling* and/or *Unmarshalling* facilities provided by Akka HTTP then the conversion of custom types to and from streamed entities can be quite convenient.

Closing a connection

The HTTP connection will be closed when the handling `Flow` cancels its upstream subscription or the peer closes the connection. An often times more convenient alternative is to explicitly add a `Connection: close` header to an `HttpResponse`. This response will then be the last one on the connection and the server will actively close the connection when it has been sent out.

2.4.4 Server-Side HTTPS Support

Akka HTTP supports TLS encryption on the server-side as well as on the *client-side*.

The central vehicle for configuring encryption is the `HttpsContext`, which is defined as such:

```
final case class HttpsContext(sslContext: SSLContext,
                              enabledCipherSuites: Option[immutable.Seq[String]] = None,
                              enabledProtocols: Option[immutable.Seq[String]] = None,
                              clientAuth: Option[ClientAuth] = None,
                              sslParameters: Option[SSLParameters] = None)
```

On the server-side the `bind`, and `bindAndHandleXXX` methods of the akka.http.scaladsl.Http extension define an optional `httpsContext` parameter, which can receive the HTTPS configuration in the form of an `HttpsContext` instance. If defined encryption is enabled on all accepted connections. Otherwise it is disabled (which is the default).

2.4.5 Stand-Alone HTTP Layer Usage

Due to its Reactive-Streams-based nature the Akka HTTP layer is fully detachable from the underlying TCP interface. While in most applications this "feature" will not be crucial it can be useful in certain cases to be able to "run" the HTTP layer (and, potentially, higher-layers) against data that do not come from the network but rather some other source. Potential scenarios where this might be useful include tests, debugging or low-level event-sourcing (e.g by replaying network traffic).

On the server-side the stand-alone HTTP layer forms a `BidiFlow` that is defined like this:

```
/**
 * The type of the server-side HTTP layer as a stand-alone BidiFlow
 * that can be put atop the TCP layer to form an HTTP server.
 *
 * {{{
 *                +------+
 * HttpResponse ~>|      |~> SslTlsOutbound
 *                | bidi |
 * HttpRequest  <~|      |<~ SslTlsInbound
 *                +------+
 * }}}
 */
type ServerLayer = BidiFlow[HttpResponse, SslTlsOutbound, SslTlsInbound, HttpRequest, Unit]
```

You create an instance of `Http.ServerLayer` by calling one of the two overloads of the `Http().serverLayer` method, which also allows for varying degrees of configuration.

2.4.6 Handling HTTP Server failures in the Low-Level API

There are various situations when failure may occur while initialising or running an Akka HTTP server. Akka by default will log all these failures, however sometimes one may want to react to failures in addition to them just being logged, for example by shutting down the actor system, or notifying some external monitoring end-point explicitly.

There are multiple things that can fail when creating and materializing an HTTP Server (similarily, the same applied to a plain streaming `Tcp()` server). The types of failures that can happen on different layers of the stack, starting from being unable to start the server, and ending with failing to unmarshal an HttpRequest, examples of failures include (from outer-most, to inner-most):

- Failure to `bind` to the specified address/port,
- Failure while accepting new `IncommingConnection` s, for example when the OS has run out of file descriptors or memory,
- Failure while handling a connection, for example if the incoming `HttpRequest` is malformed.

This section describes how to handle each failure situation, and in which situations these failures may occur.

The first type of failure is when the server is unable to bind to the given port. For example when the port is already taken by another application, or if the port is privileged (i.e. only usable by `root`). In this case the "binding future" will fail immediatly, and we can react to if by listening on the Future's completion:

```
implicit val system = ActorSystem()
implicit val materializer = ActorMaterializer()
implicit val ec = system.dispatcher

// let's say the OS won't allow us to bind to 80.
val (host, port) = ("localhost", 80)
val serverSource = Http().bind(host, port)

val bindingFuture: Future[ServerBinding] = serverSource
  .to(handleConnections) // Sink[Http.IncomingConnection, _]
  .run()

bindingFuture onFailure {
  case ex: Exception =>
    log.error(ex, "Failed to bind to {}:{}!", host, port)
}
```

Once the server has successfully bound to a port, the `Source[IncomingConnection, _]` starts running and emiting new incoming connections. This source technically can signal a failure as well, however this should only happen in very dramantic situations such as running out of file descriptors or memory available to the system, such that it's not able to accept a new incoming connection. Handling failures in Akka Streams is pretty stright forward, as failures are signaled through the stream starting from the stage which failed, all the way downstream to the final stages.

In the example below we add a custom `PushStage` (see *Custom stream processing*) in order to react to the stream's failure. We signal a `failureMonitor` actor with the cause why the stream is going down, and let the Actor handle the rest – maybe it'll decide to restart the server or shutdown the ActorSystem, that however is not our concern anymore.

```
implicit val system = ActorSystem()
implicit val materializer = ActorMaterializer()
implicit val ec = system.dispatcher

import Http._
val (host, port) = ("localhost", 8080)
val serverSource = Http().bind(host, port)

val failureMonitor: ActorRef = system.actorOf(MyExampleMonitoringActor.props)
```

```
val reactToTopLevelFailures = Flow[IncomingConnection]
  .transform { () =>
    new PushStage[IncomingConnection, IncomingConnection] {
      override def onPush(elem: IncomingConnection, ctx: Context[IncomingConnection]) =
        ctx.push(elem)

      override def onUpstreamFailure(cause: Throwable, ctx: Context[IncomingConnection]) = {
        failureMonitor ! cause
        super.onUpstreamFailure(cause, ctx)
      }
    }
  }

serverSource
  .via(reactToTopLevelFailures)
  .to(handleConnections) // Sink[Http.IncomingConnection, _]
  .run()
```

The third type of failure that can occur is when the connection has been properly established, however afterwards is terminated abruptly – for example by the client aborting the underlying TCP connection. To handle this failure we can use the same pattern as in the previous snippet, however apply it to the connection's Flow:

```
implicit val system = ActorSystem()
implicit val materializer = ActorMaterializer()
implicit val ec = system.dispatcher

val (host, port) = ("localhost", 8080)
val serverSource = Http().bind(host, port)

val reactToConnectionFailure = Flow[HttpRequest]
  .transform { () =>
    new PushStage[HttpRequest, HttpRequest] {
      override def onPush(elem: HttpRequest, ctx: Context[HttpRequest]) =
        ctx.push(elem)

      override def onUpstreamFailure(cause: Throwable, ctx: Context[HttpRequest]) = {
        // handle the failure somehow
        super.onUpstreamFailure(cause, ctx)
      }
    }
  }

val httpEcho = Flow[HttpRequest]
  .via(reactToConnectionFailure)
  .map { request =>
    // simple text "echo" response:
    HttpResponse(entity = HttpEntity(ContentTypes.`text/plain(UTF-8)`, request.entity.dataBytes))
  }

serverSource
  .runForeach { con =>
    con.handleWith(httpEcho)
  }
```

These failures can be described more or less infrastructure related, they are failing bindings or connections. Most of the time you won't need to dive into those very deeply, as Akka will simply log errors of this kind anyway, which is a reasonable default for such problems.

In order to learn more about handling exceptions in the actual routing layer, which is where your application code comes into the picture, refer to *Exception Handling* which focuses explicitly on explaining how exceptions thrown in routes can be handled and transformed into `HttpResponse` s with apropriate error codes and human-readable failure descriptions.

2.5 High-level Server-Side API

In addition to the *Low-Level Server-Side API* Akka HTTP provides a very flexible "Routing DSL" for elegantly defining RESTful web services. It picks up where the low-level API leaves off and offers much of the higher-level functionality of typical web servers or frameworks, like deconstruction of URIs, content negotiation or static content serving.

2.5.1 Routing DSL Overview

The Akka HTTP *Low-Level Server-Side API* provides a `Flow`- or `Function`-level interface that allows an application to respond to incoming HTTP requests by simply mapping requests to responses:

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.model.HttpMethods._
import akka.http.scaladsl.model._
import akka.stream.ActorMaterializer

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

val requestHandler: HttpRequest => HttpResponse = {
  case HttpRequest(GET, Uri.Path("/"), _, _, _) =>
    HttpResponse(entity = HttpEntity(ContentTypes.`text/html(UTF-8)`,
      "<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!")
}

Http().bindAndHandleSync(requestHandler, "localhost", 8080)
```

While it'd be perfectly possible to define a complete REST API service purely by pattern-matching against the incoming `HttpRequest` (maybe with the help of a few extractors in the way of Unfiltered) this approach becomes somewhat unwieldy for larger services due to the amount of syntax "ceremony" required. Also, it doesn't help in keeping your service definition as DRY as you might like.

As an alternative Akka HTTP provides a flexible DSL for expressing your service behavior as a structure of composable elements (called *Directives*) in a concise and readable way. Directives are assembled into a so called *route structure* which, at its top-level, forms a handler `Flow` (or, alternatively, an async handler function) that can be directly supplied to a `bind` call. The conversion from `Route` to flow can either be invoked explicitly using `Route.handlerFlow` or, otherwise, the conversion is also provided implicitly by `RouteResult.route2HandlerFlow`[2].

For example, the service definition from above, written using the routing DSL, would look like this:

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.marshallers.xml.ScalaXmlSupport._
import akka.http.scaladsl.server.Directives._
import akka.stream.ActorMaterializer

implicit val system = ActorSystem()
```

[2] To be picked up automatically, the implicit conversion needs to be provided in the companion object of the source type. However, as `Route` is just a type alias for `RequestContext => Future[RouteResult]`, there's no companion object for `Route`. Fortunately, the implicit scope for finding an implicit conversion also includes all types that are "associated with any part" of the source type which in this case means that the implicit conversion will also be picked up from `RouteResult.route2HandlerFlow` automatically.

```
implicit val materializer = ActorMaterializer()

val route =
  get {
    pathSingleSlash {
      complete {
        <html>
          <body>Hello world!</body>
        </html>
      }
    } ~
    path("ping") {
      complete("PONG!")
    } ~
    path("crash") {
      sys.error("BOOM!")
    }
  }

// `route` will be implicitly converted to `Flow` using `RouteResult.route2HandlerFlow`
Http().bindAndHandle(route, "localhost", 8080)
```

The core of the Routing DSL becomes available with a single import:

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

This example also relies on the pre-defined support for Scala XML with:

```
import akka.http.scaladsl.marshallers.xml.ScalaXmlSupport._
```

The very short example shown here is certainly not the best for illustrating the savings in "ceremony" and improvements in conciseness and readability that the Routing DSL promises. The *Longer Example* might do a better job in this regard.

For learning how to work with the Routing DSL you should first understand the concept of *Routes*.

2.5.2 Routes

The "Route" is the central concept of Akka HTTP's Routing DSL. All the structures you build with the DSL, no matter whether they consists of a single line or span several hundred lines, are instances of this type:

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

It's a simple alias for a function turning a `RequestContext` into 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

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`, which relies on the in-scope `RejectionHandler` and `ExceptionHandler` instances to convert rejections and exceptions into appropriate HTTP responses for the client.

Using `Route.handlerFlow` or `Route.asyncHandler` a `Route` can be lifted into a handler `Flow` or async handler function to be used with a `bindAndHandleXXX` call from the *Low-Level Server-Side API*.

Note: There is also an implicit conversion from `Route` to `Flow[HttpRequest, HttpResponse, Unit]` defined in the `RouteResult` companion, which relies on `Route.handlerFlow`.

RequestContext

The request context wraps an `HttpRequest` instance to enrich it with additional information that are typically required by the routing logic, like an `ExecutionContext`, `Materializer`, `LoggingAdapter` and the configured `RoutingSettings`. It also contains the `unmatchedPath`, a value that describes how much of the request URI has not yet been matched by a *Path Directive*.

The `RequestContext` itself is immutable but contains several helper methods which allow for convenient creation of modified copies.

RouteResult

`RouteResult` is a simple abstract data type (ADT) that models the possible non-error results of a `Route`. It is defined as such:

```
sealed trait RouteResult

object RouteResult {
  final case class Complete(response: HttpResponse) extends RouteResult
  final case class Rejected(rejections: immutable.Seq[Rejection]) extends RouteResult
}
```

Usually you don't create any `RouteResult` instances yourself, but rather rely on the pre-defined *RouteDirectives* (like *complete*, *reject* or *redirect*) or the respective methods on the *RequestContext* instead.

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 that becomes available when you `import akka.http.scaladsl.server.Directives._`. 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.

2.5.3 Directives

A "Directive" is a small building block used for creating arbitrarily complex *route structures*. Akka HTTP already pre-defines a large number of directives and you can easily construct your own:

Predefined Directives (alphabetically)

Directive	Description
authenticateBasic	Wraps the inner route with Http Basic authentication support using a given `Authenti`
authenticateBasicAsync	Wraps the inner route with Http Basic authentication support using a given `AsyncAut`
authenticateBasicPF	Wraps the inner route with Http Basic authentication support using a given `Authenti`
authenticateBasicPFAsync	Wraps the inner route with Http Basic authentication support using a given `AsyncAut`
authenticateOAuth2	Wraps the inner route with OAuth Bearer Token authentication support using a given `Au`
authenticateOAuth2Async	Wraps the inner route with OAuth Bearer Token authentication support using a given `As`
authenticateOAuth2PF	Wraps the inner route with OAuth Bearer Token authentication support using a given `Au`
authenticateOAuth2PFAsync	Wraps the inner route with OAuth Bearer Token authentication support using a given `As`
authenticateOrRejectWithChallenge	Lifts an authenticator function into a directive
authorize	Applies the given authorization check to the request
cancelRejection	Adds a `TransformationRejection` cancelling all rejections equal to the given on
cancelRejections	Adds a `TransformationRejection` cancelling all matching rejections to the rejec
complete	Completes the request using the given arguments
completeOrRecoverWith	"Unwraps" a `Future[T]` and runs the inner route when the future has failed with the
completeWith	Uses the marshaller for a given type to extract a completion function
conditional	Wraps its inner route with support for conditional requests as defined by http://tools.ietf
cookie	Extracts the `HttpCookie` with the given name
decodeRequest	Decompresses the request if it is `gzip` or `deflate` compressed
decodeRequestWith	Decodes the incoming request using one of the given decoders
delete	Rejects all non-DELETE requests
deleteCookie	Adds a `Set-Cookie` response header expiring the given cookies
encodeResponse	Encodes the response with the encoding that is requested by the client via the `Accept-`
encodeResponseWith	Encodes the response with the encoding that is requested by the client via the `Accept-`
entity	Extracts the request entity unmarshalled to a given type
extract	Extracts a single value using a `RequestContext ⇒ T` function
extractClientIP	Extracts the client's IP from either the `X-Forwarded-`, `Remote-Address` or `X-Re`
extractCredentials	Extracts the potentially present `HttpCredentials` provided with the request's `Auth`

Table 2.1 – con

Directive	Description
extractExecutionContext	Extracts the `ExecutionContext` from the `RequestContext`
extractMaterializer	Extracts the `Materializer` from the `RequestContext`
extractHost	Extracts the hostname part of the Host request header value
extractLog	Extracts the `LoggingAdapter` from the `RequestContext`
extractMethod	Extracts the request method
extractRequest	Extracts the current `HttpRequest` instance
extractRequestContext	Extracts the `RequestContext` itself
extractScheme	Extracts the URI scheme from the request
extractSettings	Extracts the `RoutingSettings` from the `RequestContext`
extractUnmatchedPath	Extracts the yet unmatched path from the `RequestContext`
extractUri	Extracts the complete request URI
failWith	Bubbles the given error up the response chain where it is dealt with by the closest *handl*
fileUpload	Provides a stream of an uploaded file from a multipart request
formField	Extracts an HTTP form field from the request
formFields	Extracts a number of HTTP form field from the request
get	Rejects all non-GET requests
getFromBrowseableDirectories	Serves the content of the given directories as a file-system browser, i.e. files are sent and
getFromBrowseableDirectory	Serves the content of the given directory as a file-system browser, i.e. files are sent and
getFromDirectory	Completes GET requests with the content of a file underneath a given file-system direct
getFromFile	Completes GET requests with the content of a given file
getFromResource	Completes GET requests with the content of a given class-path resource
getFromResourceDirectory	Completes GET requests with the content of a file underneath a given "class-path resour
handleExceptions	Transforms exceptions thrown during evaluation of the inner route using the given `Exc`
handleRejections	Transforms rejections produced by the inner route using the given `RejectionHandl`
handleWebsocketMessages	Handles websocket requests with the given handler and rejects other requests with an `E`
handleWebsocketMessagesForProtocol	Handles websocket requests with the given handler if the subprotocol matches and rejec
handleWith	Completes the request using a given function
head	Rejects all non-HEAD requests
headerValue	Extracts an HTTP header value using a given `HttpHeader ⇒ Option[T]` functio
headerValueByName	Extracts the value of the first HTTP request header with a given name
headerValueByType	Extracts the first HTTP request header of the given type
headerValuePF	Extracts an HTTP header value using a given `PartialFunction[HttpHeader,`
host	Rejects all requests with a non-matching host name
listDirectoryContents	Completes GET requests with a unified listing of the contents of all given file-system di
logRequest	Produces a log entry for every incoming request
logRequestResult	Produces a log entry for every incoming request and `RouteResult`
logResult	Produces a log entry for every `RouteResult`
mapInnerRoute	Transforms its inner `Route` with a `Route => Route` function
mapRejections	Transforms rejections from a previous route with an `immutable.Seq[Rejection`
mapRequest	Transforms the request with an `HttpRequest => HttpRequest` function
mapRequestContext	Transforms the `RequestContext` with a `RequestContext => RequestCont`
mapResponse	Transforms the response with an `HttpResponse => HttpResponse` function
mapResponseEntity	Transforms the response entity with an `ResponseEntity ⇒ ResponseEntity`
mapResponseHeaders	Transforms the response headers with an `immutable.Seq[HttpHeader] ⇒ im`
mapRouteResult	Transforms the `RouteResult` with a `RouteResult ⇒ RouteResult` function
mapRouteResultFuture	Transforms the `RouteResult` future with a `Future[RouteResult] ⇒ Futur`
mapRouteResultPF	Transforms the `RouteResult` with a `PartialFunction[RouteResult, Rou`
mapRouteResultWith	Transforms the `RouteResult` with a `RouteResult ⇒ Future[RouteResult`
mapRouteResultWithPF	Transforms the `RouteResult` with a `PartialFunction[RouteResult, Fut`
mapSettings	Transforms the `RoutingSettings` with a `RoutingSettings ⇒ RoutingSet`
mapUnmatchedPath	Transforms the `unmatchedPath` of the `RequestContext` using a `Uri.Path ⇒`
method	Rejects all requests whose HTTP method does not match the given one
onComplete	"Unwraps" a `Future[T]` and runs the inner route after future completion with the fut
onSuccess	"Unwraps" a `Future[T]` and runs the inner route after future completion with the fut
optionalCookie	Extracts the `HttpCookiePair` with the given name as an `Option[HttpCookieP`

Table 2.1 – con

Directive	Description
optionalHeaderValue	Extracts an optional HTTP header value using a given `HttpHeader ⇒ Option[T`
optionalHeaderValueByName	Extracts the value of the first optional HTTP request header with a given name
optionalHeaderValueByType	Extracts the first optional HTTP request header of the given type
optionalHeaderValuePF	Extracts an optional HTTP header value using a given `PartialFunction[HttpHe`
options	Rejects all non-OPTIONS requests
overrideMethodWithParameter	Changes the request method to the value of the specified query parameter
parameter	Extracts a query parameter value from the request
parameterMap	Extracts the request's query parameters as a `Map[String, String]`
parameterMultiMap	Extracts the request's query parameters as a `Map[String, List[String]]`
parameters	Extracts a number of query parameter values from the request
parameterSeq	Extracts the request's query parameters as a `Seq[(String, String)]`
pass	Always simply passes the request on to its inner route, i.e. doesn't do anything, neither
patch	Rejects all non-PATCH requests
path	Applies the given `PathMatcher` to the remaining unmatched path after consuming a
pathEnd	Only passes on the request to its inner route if the request path has been matched compl
pathEndOrSingleSlash	Only passes on the request to its inner route if the request path has been matched compl
pathPrefix	Applies the given `PathMatcher` to a prefix of the remaining unmatched path after co
pathPrefixTest	Checks whether the unmatchedPath has a prefix matched by the given `PathMatcher`
pathSingleSlash	Only passes on the request to its inner route if the request path consists of exactly one re
pathSuffix	Applies the given `PathMatcher` to a suffix of the remaining unmatched path (Caution
pathSuffixTest	Checks whether the unmatched path has a suffix matched by the given `PathMatcher`
post	Rejects all non-POST requests
provide	Injects a given value into a directive
put	Rejects all non-PUT requests
rawPathPrefix	Applies the given matcher directly to a prefix of the unmatched path of the `RequestC`
rawPathPrefixTest	Checks whether the unmatchedPath has a prefix matched by the given `PathMatcher`
recoverRejections	Transforms rejections from the inner route with an `immutable.Seq[Rejection]`
recoverRejectionsWith	Transforms rejections from the inner route with an `immutable.Seq[Rejection]`
redirect	Completes the request with redirection response of the given type to the given URI
redirectToNoTrailingSlashIfPresent	If the request path ends with a slash, redirects to the same uri without trailing slash in th
redirectToTrailingSlashIfMissing	If the request path doesn't end with a slash, redirects to the same uri with trailing slash i
reject	Rejects the request with the given rejections
rejectEmptyResponse	Converts responses with an empty entity into (empty) rejections
requestEncodedWith	Rejects the request with an `UnsupportedRequestEncodingRejection` if its en
requestEntityEmpty	Rejects if the request entity is non-empty
requestEntityPresent	Rejects with a `RequestEntityExpectedRejection` if the request entity is empt
respondWithDefaultHeader	Adds a given response header if the response doesn't already contain a header with the s
respondWithDefaultHeaders	Adds the subset of the given headers to the response which doesn't already have a heade
respondWithHeader	Unconditionally adds a given header to the outgoing response
respondWithHeaders	Unconditionally adds the given headers to the outgoing response
responseEncodingAccepted	Rejects the request with an `UnacceptedResponseEncodingRejection` if the g
scheme	Rejects all requests whose URI scheme doesn't match the given one
selectPreferredLanguage	Inspects the request's `Accept-Language` header and determines, which of a given se
setCookie	Adds a `Set-Cookie` response header with the given cookies
textract	Extracts a number of values using a `RequestContext ⇒ Tuple` function
tprovide	Injects a given tuple of values into a directive
uploadedFile	Streams one uploaded file from a multipart request to a file on disk
validate	Checks a given condition before running its inner route
withExecutionContext	Runs its inner route with the given alternative `ExecutionContext`
withMaterializer	Runs its inner route with the given alternative `Materializer`
withLog	Runs its inner route with the given alternative `LoggingAdapter`
withRangeSupport	Adds `Accept-Ranges: bytes` to responses to GET requests, produces partial res
withSettings	Runs its inner route with the given alternative `RoutingSettings`

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.

Marshalling Directives 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.

FileUploadDirectives Handle file uploads.

Directives creating or transforming the response

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

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.

Providing Values to Inner Routes These directives provide values to 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 a tuple of values.

- *extract*
- *extractExecutionContext*
- *extractMaterializer*
- *extractLog*
- *extractRequest*

- *extractRequestContext*
- *extractSettings*
- *extractUnmatchedPath*
- *extractUri*
- *textract*
- *provide*
- *tprovide*

Transforming the Request(Context)

- *mapRequest*
- *mapRequestContext*
- *mapSettings*
- *mapUnmatchedPath*
- *withExecutionContext*
- *withMaterializer*
- *withLog*
- *withSettings*

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*

Transforming the RouteResult These directives allow to transform the RouteResult of the inner route.

- *cancelRejection*
- *cancelRejections*
- *mapRejections*
- *mapRouteResult*
- *mapRouteResultFuture*
- *mapRouteResultPF*
- *mapRouteResultWith*
- *mapRouteResultWithPF*
- *recoverRejections*
- *recoverRejectionsWith*

Other

- *mapInnerRoute*
- *pass*

Alphabetically

cancelRejection

Signature

```
def cancelRejection(rejection: Rejection): Directive0
```

Description Adds a `TransformationRejection` cancelling all rejections equal to the given one to the rejections potentially coming back from the inner route.

Read *Rejections* to learn more about rejections.

For more advanced handling of rejections refer to the *handleRejections* directive which provides a nicer DSL for building rejection handlers.

Example

```
val route =
  cancelRejection(MethodRejection(HttpMethods.POST)) {
    post {
      complete("Result")
    }
  }

// tests:
Get("/") ~> route ~> check {
  rejections shouldEqual Nil
  handled shouldEqual false
}
```

cancelRejections

Signature

```
def cancelRejections(classes: Class[_]*): Directive0
def cancelRejections(cancelFilter: Rejection ⇒ Boolean): Directive0
```

Description Adds a `TransformationRejection` cancelling all rejections created by the inner route for which the condition argument function returns `true`.

See also *cancelRejection*, for canceling a specific rejection.

Read *Rejections* to learn more about rejections.

For more advanced handling of rejections refer to the *handleRejections* directive which provides a nicer DSL for building rejection handlers.

Example

```
def isMethodRejection: Rejection => Boolean = {
  case MethodRejection(_) => true
  case _                  => false
}

val route =
  cancelRejections(isMethodRejection) {
    post {
      complete("Result")
    }
```

```
  }

// tests:
Get("/") ~> route ~> check {
  rejections shouldEqual Nil
  handled shouldEqual false
}
```

extract

Signature

```
def extract[T](f: RequestContext ⇒ T): Directive1[T]
```

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 *Providing 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")
  }

// tests:
Get("/abcdef") ~> route ~> check {
  responseAs[String] shouldEqual "The length of the request URI is 25"
}
val route =
  extractLog { log =>
    log.debug("I'm logging things in much detail..!")
    complete("It's amazing!")
  }

// tests:
Get("/abcdef") ~> route ~> check {
  responseAs[String] shouldEqual "It's amazing!"
}
```

extractExecutionContext

Signature

```
def extractExecutionContext: Directive1[ExecutionContext]
```

Description Extracts the `ExecutionContext` from the `RequestContext`.

See *withExecutionContext* to see how to customise the execution context provided for an inner route.

See *extract* to learn more about how extractions work.

Example

```
def sample() =
  path("sample") {
    extractExecutionContext { implicit ec =>
      complete {
        Future(s"Run on ${ec.##}!") // uses the `ec` ExecutionContext
      }
    }
  }

val route =
  pathPrefix("special") {
    sample() // default execution context will be used
  }

// tests:
Get("/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"Run on ${system.dispatcher.##}!"
}
```

extractMaterializer

Signature

```
def extractMaterializer: Directive1[Materializer]
```

Description Extracts the `Materializer` from the `RequestContext`, which can be useful when you want to run an Akka Stream directly in your route.

See also *withMaterializer* to see how to customise the used materializer for specific inner routes.

Example

```
val route =
  path("sample") {
    extractMaterializer { materializer =>
      complete {
        // explicitly use the `materializer`:
        Source.single(s"Materialized by ${materializer.##}!")
          .runWith(Sink.head)(materializer)
      }
    }
  } // default materializer will be used

// tests:
Get("/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"Materialized by ${materializer.##}!"
}
```

extractLog

Signature

```
def extractLog: Directive1[LoggingAdapter]
```

Description Extracts a `LoggingAdapter` from the request context which can be used for logging inside the route.

The `extractLog` directive is used for providing logging to routes, such that they don't have to depend on closing over a logger provided in the class body.

See *extract* and *Providing Values to Inner Routes* for an overview of similar directives.

Example

```
val route =
  extractLog { log =>
    log.debug("I'm logging things in much detail..!")
    complete("It's amazing!")
  }

// tests:
Get("/abcdef") ~> route ~> check {
  responseAs[String] shouldEqual "It's amazing!"
}
```

extractRequest

Signature

```
def extractRequest: Directive1[HttpRequest]
```

Description Extracts the complete `HttpRequest` instance.

Use `extractRequest` to extract just the complete URI of the request. Usually there's little use of extracting the complete request because extracting of most of the aspects of HttpRequests is handled by specialized directives. See *Directives filtering or extracting from the request*.

Example

```
val route =
  extractRequest { request =>
    complete(s"Request method is ${request.method.name} and content-type is ${request.entity.conte
  }

// tests:
Post("/", "text") ~> route ~> check {
  responseAs[String] shouldEqual "Request method is POST and content-type is text/plain; charset=U
}
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "Request method is GET and content-type is none/none"
}
```

extractRequestContext

Signature

```
def extractRequestContext: Directive1[RequestContext]
```

Description Extracts the request's underlying `RequestContext`.

This directive is used as a building block for most of the other directives, which extract the context and by inspecting some of it's values can decide what to do with the request - for example provide a value, or reject the request.

See also *extractRequest* if only interested in the `HttpRequest` instance itself.

Example

```
val route =
  extractRequestContext { ctx =>
    ctx.log.debug("Using access to additional context availablethings, like the logger.")
    val request = ctx.request
    complete(s"Request method is ${request.method.name} and content-type is ${request.entity.conte
  }

// tests:
Post("/", "text") ~> route ~> check {
  responseAs[String] shouldEqual "Request method is POST and content-type is text/plain; charset=U
}
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "Request method is GET and content-type is none/none"
}
```

extractSettings

Signature

```
def extractSettings: Directive1[RoutingSettings]
```

Description Extracts the `RoutingSettings` from the `RequestContext`.

By default the settings of the `Http()` extension running the route will be returned. It is possible to override the settings for specific sub-routes by using the *withSettings* directive.

Example

```
val route =
  extractSettings { settings: RoutingSettings =>
    complete(s"RoutingSettings.renderVanityFooter = ${settings.renderVanityFooter}")
  }

// tests:
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual s"RoutingSettings.renderVanityFooter = true"
}
```

extractUnmatchedPath

Signature

```
def extractUnmatchedPath: Directive1[Uri.Path]
```

Description Extracts the unmatched path from the request context.

The `extractUnmatchedPath` directive extracts the remaining path that was not yet matched by any of the *PathDirectives* (or any custom ones that change the unmatched path field of the request context). You can use it for building directives that handle complete suffixes of paths (like the `getFromDirectory` directives and similar ones).

Use `mapUnmatchedPath` to change the value of the unmatched path.

Example

```
val route =
  pathPrefix("abc") {
    extractUnmatchedPath { remaining =>
      complete(s"Unmatched: '$remaining'")
    }
  }

// tests:
Get("/abc") ~> route ~> check {
  responseAs[String] shouldEqual "Unmatched: ''"
}
Get("/abc/456") ~> route ~> check {
  responseAs[String] shouldEqual "Unmatched: '/456'"
}
```

extractUri

Signature

```
def extractUri: Directive1[Uri]
```

Description Access the full URI of the request.

Use *SchemeDirectives*, *HostDirectives*, *PathDirectives*, and *ParameterDirectives* for more targeted access to parts of the URI.

Example

```
val route =
  extractUri { uri =>
    complete(s"Full URI: $uri")
  }

// tests:
Get("/") ~> route ~> check {
  // tests are executed with the host assumed to be "example.com"
  responseAs[String] shouldEqual "Full URI: http://example.com/"
}
Get("/test") ~> route ~> check {
  responseAs[String] shouldEqual "Full URI: http://example.com/test"
}
```

mapInnerRoute

Signature

```
def mapInnerRoute(f: Route ⇒ Route): Directive0
```

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

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"))
  }

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

mapRejections

Signature

```
def mapRejections(f: immutable.Seq[Rejection] ⇒ immutable.Seq[Rejection]): Directive0
```

Description **Low level directive** – unless you're sure you need to be working on this low-level you might instead want to try the *handleRejections* directive which provides a nicer DSL for building rejection handlers.

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 *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"))
  }

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

Get("/abc") ~> route ~> check {
  status shouldEqual StatusCodes.OK
}
```

mapRequest

Signature

```
def mapRequest(f: HttpRequest ⇒ HttpRequest): Directive0
```

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

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 *Transforming the Request(Context)* 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

Signature

```
def mapRequestContext(f: RequestContext ⇒ RequestContext): Directive0
```

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

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 *Transforming the Request(Context)* for an overview of similar directives.

Example

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

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

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

mapResponse

Signature

```
def mapResponse(f: HttpResponse ⇒ HttpResponse): Directive0
```

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 complete responses.

See also *mapResponseHeaders* or *mapResponseEntity* for more specialized variants and *Transforming the Response* for similar directives.

Example: Override status

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

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

Example: Default to empty JSON response on errors

```
trait ApiRoutes {
  protected def system: ActorSystem
  private val log = Logging(system, "ApiRoutes")

  private val NullJsonEntity = HttpEntity(ContentTypes.`application/json`, "{}")

  private def nonSuccessToEmptyJsonEntity(response: HttpResponse): HttpResponse =
    response.status match {
      case code if code.isSuccess ⇒ response
      case code ⇒
        log.warning("Dropping response entity since response status code was: {}", code)
        response.copy(entity = NullJsonEntity)
    }

  /** Wrapper for all of our JSON API routes */
  def apiRoute(innerRoutes: ⇒ Route): Route =
    mapResponse(nonSuccessToEmptyJsonEntity)(innerRoutes)
}
val route: Route =
  apiRoute {
    get {
      complete(InternalServerError)
    }
  }

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

mapResponseEntity

Signature

```
def mapResponseEntity(f: ResponseEntity ⇒ ResponseEntity): Directive0
```

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 *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"))

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

mapResponseHeaders

Signature

```
def mapResponseHeaders(f: immutable.Seq[HttpHeader] ⇒ immutable.Seq[HttpHeader]): Directive0
```

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

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 *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")
    }
  }

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

mapRouteResult

Signature

```
def mapRouteResult(f: RouteResult ⇒ RouteResult): Directive0
```

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

The `mapRouteResult` directive is used as a building block for *Custom Directives* to transform the *RouteResult* coming back from the inner route.

See *Transforming the RouteResult* for similar directives.

Example

```
val rejectAll = // not particularly useful directive
  mapRouteResult {
    case _ => Rejected(List(AuthorizationFailedRejection))
  }
val route =
  rejectAll {
    complete("abc")
  }

// tests:
Get("/") ~> route ~> check {
  rejections.nonEmpty shouldEqual true
}
```

mapRouteResultFuture

Signature

```
def mapRouteResultFuture(f: Future[RouteResult] ⇒ Future[RouteResult]): Directive0
```

Description Asynchronous version of *mapRouteResult*.

It's similar to *mapRouteResultWith*, however it's `Future[RouteResult] ⇒ Future[RouteResult]` instead of `RouteResult ⇒ Future[RouteResult]` which may be useful when combining multiple transformantions and / or wanting to `recover` from a failed route result.

See *Transforming the RouteResult* for similar directives.

Example

```
val tryRecoverAddServer = mapRouteResultFuture { fr =>
  fr recover {
    case ex: IllegalArgumentException =>
      Complete(HttpResponse(StatusCodes.InternalServerError))
  } map {
    case Complete(res) => Complete(res.addHeader(Server("MyServer 1.0")))
    case rest          => rest
  }
}

val route =
  tryRecoverAddServer {
    complete("Hello world!")
  }

// tests:
Get("/") ~> route ~> check {
  status shouldEqual StatusCodes.OK
  header[Server] shouldEqual Some(Server("MyServer 1.0"))
}
```

mapRouteResultPF

Signature

```
def mapRouteResultPF(f: PartialFunction[RouteResult, RouteResult]): Directive0
```

Description *Partial Function* version of *mapRouteResult*.

Changes the message the inner route sends to the responder.

The `mapRouteResult` directive is used as a building block for *Custom Directives* to transform the *RouteResult* coming back from the inner route. It's similar to the *mapRouteResult* directive but allows to specify a partial function that doesn't have to handle all potential `RouteResult` instances.

See *Transforming the RouteResult* for similar directives.

Example

```
case object MyCustomRejection extends Rejection
val rejectRejections = // not particularly useful directive
  mapRouteResultPF {
    case Rejected(_) => Rejected(List(AuthorizationFailedRejection))
  }
val route =
  rejectRejections {
    reject(MyCustomRejection)
  }

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

mapRouteResultWith

Signature

```
def mapRouteResultWith(f: RouteResult ⇒ Future[RouteResult]): Directive0
```

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

The `mapRouteResult` directive is used as a building block for *Custom Directives* to transform the *RouteResult* coming back from the inner route. It's similar to the *mapRouteResult* directive but returning a `Future` instead of a result immediadly, which may be useful for longer running transformations.

See *Transforming the RouteResult* for similar directives.

Example

```
case object MyCustomRejection extends Rejection
val rejectRejections = // not particularly useful directive
  mapRouteResultWith { res =>
    res match {
      case Rejected(_) => Future(Rejected(List(AuthorizationFailedRejection)))
      case _           => Future(res)
    }
  }
val route =
  rejectRejections {
    reject(MyCustomRejection)
  }

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

mapRouteResultWithPF

Signature

```
def mapRouteResultWithPF(f: PartialFunction[RouteResult, Future[RouteResult]]): Directive0
```

Description Asynchronous variant of *mapRouteResultPF*.

Changes the message the inner route sends to the responder.

The `mapRouteResult` directive is used as a building block for *Custom Directives* to transform the *RouteResult* coming back from the inner route.

See *Transforming the RouteResult* for similar directives.

Example

```
case object MyCustomRejection extends Rejection
val rejectRejections = // not particularly useful directive
  mapRouteResultWithPF {
    case Rejected(_) => Future(Rejected(List(AuthorizationFailedRejection)))
  }
val route =
  rejectRejections {
    reject(MyCustomRejection)
  }

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

mapSettings

Signature

```
def mapSettings(f: RoutingSettings ⇒ RoutingSettings): Directive0
```

Description Transforms the `RoutingSettings` with a `RoutingSettings ⇒ RoutingSettings` function.

See also *withSettings* or *extractSettings*.

Example

```
val special = RoutingSettings(system).copy(fileIODispatcher = "special-io-dispatcher")

def sample() =
  path("sample") {
    complete {
      // internally uses the configured fileIODispatcher:
      val source = FileIO.fromFile(new File("example.json"))
      HttpResponse(entity = HttpEntity(ContentTypes.`application/json`, source))
    }
  }

val route =
  get {
    pathPrefix("special") {
      withSettings(special) {
```

```
        sample() // `special` file-io-dispatcher will be used to read the file
      }
    } ~ sample() // default file-io-dispatcher will be used to read the file
  }

// tests:
Post("/special/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"{}"
}
Get("/sample") ~> route ~> check {
  responseAs[String] shouldEqual "{}"
}
```

mapUnmatchedPath

Signature

```
def mapUnmatchedPath(f: Uri.Path ⇒ Uri.Path): Directive0
```

Description Transforms the unmatchedPath field of the request context for inner routes.

The `mapUnmatchedPath` directive is used as a building block for writing *Custom Directives*. You can use it for implementing custom path matching directives.

Use `extractUnmatchedPath` for extracting the current value of the unmatched path.

Example

```
def ignore456(path: Uri.Path) = path match {
  case s @ Uri.Path.Segment(head, tail) if head.startsWith("456") =>
    val newHead = head.drop(3)
    if (newHead.isEmpty) tail
    else s.copy(head = head.drop(3))
  case _ => path
}
val ignoring456 = mapUnmatchedPath(ignore456)

val route =
  pathPrefix("123") {
    ignoring456 {
      path("abc") {
        complete(s"Content")
      }
    }
  }

// tests:
Get("/123/abc") ~> route ~> check {
  responseAs[String] shouldEqual "Content"
}
Get("/123456/abc") ~> route ~> check {
  responseAs[String] shouldEqual "Content"
}
```

pass

Signature

```
def pass: Directive0
```

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

It is usually used as a "neutral element" when combining directives generically.

Example

```
val route = pass(complete("abc"))

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

provide

Signature

```
def provide[T](value: T): Directive1[T]
```

Description Provides a constant value to the inner route.

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 *Providing 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)
  }

// tests:
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "prefix:test"
}
```

recoverRejections

Signature

```
def recoverRejections(f: immutable.Seq[Rejection] ⇒ RouteResult): Directive0
```

Description **Low level directive** – unless you're sure you need to be working on this low-level you might instead want to try the *handleRejections* directive which provides a nicer DSL for building rejection handlers.

Transforms rejections from the inner route with an `immutable.Seq[Rejection] ⇒ RouteResult` function. A `RouteResult` is either a `Complete(HttpResponse(...))` or rejections `Rejected(rejections)`.

Note: To learn more about how and why rejections work read the *Rejections* section of the documentation.

Example

```
val authRejectionsToNothingToSeeHere = recoverRejections { rejections =>
  if (rejections.exists(_.isInstanceOf[AuthenticationFailedRejection]))
    Complete(HttpResponse(entity = "Nothing to see here, move along."))
  else if (rejections == Nil) // see "Empty Rejections" for more details
    Complete(HttpResponse(StatusCodes.NotFound, entity = "Literally nothing to see here."))
  else
    Rejected(rejections)
}
val neverAuth: Authenticator[String] = creds => None
val alwaysAuth: Authenticator[String] = creds => Some("id")

val route =
  authRejectionsToNothingToSeeHere {
    pathPrefix("auth") {
      path("never") {
        authenticateBasic("my-realm", neverAuth) { user =>
          complete("Welcome to the bat-cave!")
        }
      } ~
        path("always") {
          authenticateBasic("my-realm", alwaysAuth) { user =>
            complete("Welcome to the secret place!")
          }
        }
    }
  }

// tests:
Get("/auth/never") ~> route ~> check {
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "Nothing to see here, move along."
}
Get("/auth/always") ~> route ~> check {
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "Welcome to the secret place!"
}
Get("/auth/does_not_exist") ~> route ~> check {
  status shouldEqual StatusCodes.NotFound
  responseAs[String] shouldEqual "Literally nothing to see here."
}
val authRejectionsToNothingToSeeHere = recoverRejectionsWith { rejections =>
  Future {
    // imagine checking rejections takes a longer time:
    if (rejections.exists(_.isInstanceOf[AuthenticationFailedRejection]))
      Complete(HttpResponse(entity = "Nothing to see here, move along."))
    else
      Rejected(rejections)
  }
}
val neverAuth: Authenticator[String] = creds => None

val route =
  authRejectionsToNothingToSeeHere {
    pathPrefix("auth") {
      path("never") {
        authenticateBasic("my-realm", neverAuth) { user =>
          complete("Welcome to the bat-cave!")
        }
      }
    }
  }
```

```
// tests:
Get("/auth/never") ~> route ~> check {
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "Nothing to see here, move along."
}
```

recoverRejectionsWith

Signature

```
def recoverRejectionsWith(f: immutable.Seq[Rejection] ⇒ Future[RouteResult]): Directive0
```

Description **Low level directive** – unless you're sure you need to be working on this low-level you might instead want to try the *handleRejections* directive which provides a nicer DSL for building rejection handlers.

Transforms rejections from the inner route with an `immutable.Seq[Rejection] ⇒ Future[RouteResult]` function.

Asynchronous version of *recoverRejections*.

See *recoverRejections* (the synchronous equivalent of this directive) for a detailed description.

Note: To learn more about how and why rejections work read the *Rejections* section of the documentation.

Example

```
val authRejectionsToNothingToSeeHere = recoverRejectionsWith { rejections =>
  Future {
    // imagine checking rejections takes a longer time:
    if (rejections.exists(_.isInstanceOf[AuthenticationFailedRejection]))
      Complete(HttpResponse(entity = "Nothing to see here, move along."))
    else
      Rejected(rejections)
  }
}
val neverAuth: Authenticator[String] = creds => None

val route =
  authRejectionsToNothingToSeeHere {
    pathPrefix("auth") {
      path("never") {
        authenticateBasic("my-realm", neverAuth) { user =>
          complete("Welcome to the bat-cave!")
        }
      }
    }
  }

// tests:
Get("/auth/never") ~> route ~> check {
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "Nothing to see here, move along."
}
```

textract

Signature

```
def textract[L: Tuple](f: RequestContext ⇒ L): Directive[L]
```

Description Extracts a tuple of values from the request context and provides them to the inner route.

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 *Providing Values to Inner Routes* for an overview of similar directives.

See also *extract* for extracting a single value.

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")
  }

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

tprovide

Signature

```
def tprovide[L: Tuple](values: L): Directive[L]
```

Description Provides a tuple of values to the inner route.

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 *Providing Values to Inner Routes* for an overview of similar directives.

See also *provide* for providing a single value.

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")
  }

// tests:
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "Value is test and its length is 4"
}
```

withExecutionContext

Signature

```
def withExecutionContext(ec: ExecutionContext): Directive0
```

Description Allows running an inner route using an alternative `ExecutionContext` in place of the default one.

The execution context can be extracted in an inner route using *extractExecutionContext* directly, or used by directives which internally extract the materializer without sufracing this fact in the API.

Example

```
val special = system.dispatchers.lookup("special")

def sample() =
  path("sample") {
    extractExecutionContext { implicit ec =>
      complete {
        Future(s"Run on ${ec.##}!") // uses the `ec` ExecutionContext
      }
    }
  }

val route =
  pathPrefix("special") {
    withExecutionContext(special) {
      sample() // `special` execution context will be used
    }
  } ~ sample() // default execution context will be used

// tests:
Get("/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"Run on ${system.dispatcher.##}!"
}
Get("/special/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"Run on ${special.##}!"
}
```

withMaterializer

Signature

```
def withMaterializer(materializer: Materializer): Directive0
```

Description Allows running an inner route using an alternative `Materializer` in place of the default one.

The materializer can be extracted in an inner route using *extractMaterializer* directly, or used by directives which internally extract the materializer without sufracing this fact in the API (e.g. responding with a Chunked entity).

Example

```
val special = ActorMaterializer(namePrefix = Some("special"))

def sample() =
  path("sample") {
    extractMaterializer { mat =>
      complete {
        // explicitly use the materializer:
        Source.single(s"Materialized by ${mat.##}!")
          .runWith(Sink.head)(mat)
```

```
      }
    }
  }

val route =
  pathPrefix("special") {
    withMaterializer(special) {
      sample() // `special` materializer will be used
    }
  } ~ sample() // default materializer will be used

// tests:
Get("/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"Materialized by ${materializer.##}!"
}
Get("/special/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"Materialized by ${special.##}!"
}
```

withLog

Signature

```
def withLog(log: LoggingAdapter): Directive0
```

Description Allows running an inner route using an alternative `LoggingAdapter` in place of the default one.

The logging adapter can be extracted in an inner route using *extractLog* directly, or used by directives which internally extract the materializer without sufracing this fact in the API.

Example

```
val special = Logging(system, "SpecialRoutes")

def sample() =
  path("sample") {
    extractLog { implicit log =>
      complete {
        val msg = s"Logging using $log!"
        log.debug(msg)
        msg
      }
    }
  }

val route =
  pathPrefix("special") {
    withLog(special) {
      sample() // `special` logging adapter will be used
    }
  } ~ sample() // default logging adapter will be used

// tests:
Get("/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"Logging using ${system.log}!"
}
Get("/special/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"Logging using $special!"
}
```

withSettings

Signature

```
def withSettings(settings: RoutingSettings): Directive0
```

Description Allows running an inner route using an alternative `RoutingSettings` in place of the default one.

The execution context can be extracted in an inner route using *extractSettings* directly, or used by directives which internally extract the materializer without sufracing this fact in the API.

Example

```
val special = RoutingSettings(system).copy(fileIODispatcher = "special-io-dispatcher")

def sample() =
  path("sample") {
    complete {
      // internally uses the configured fileIODispatcher:
      val source = FileIO.fromFile(new File("example.json"))
      HttpResponse(entity = HttpEntity(ContentTypes.`application/json`, source))
    }
  }

val route =
  get {
    pathPrefix("special") {
      withSettings(special) {
        sample() // `special` file-io-dispatcher will be used to read the file
      }
    } ~ sample() // default file-io-dispatcher will be used to read the file
  }

// tests:
Post("/special/sample") ~> route ~> check {
  responseAs[String] shouldEqual s"{}"
}
Get("/sample") ~> route ~> check {
  responseAs[String] shouldEqual "{}"
}
```

CacheConditionDirectives

conditional

Signature

```
def conditional(eTag: EntityTag): Directive0
def conditional(lastModified: DateTime): Directive0
def conditional(eTag: EntityTag, lastModified: DateTime): Directive0
def conditional(eTag: Option[EntityTag], lastModified: Option[DateTime]): Directive0
```

Description Wraps its inner route with support for Conditional Requests as defined by http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-26.

Depending on the given `eTag` and `lastModified` values this directive immediately responds with `304 Not Modified` or `412 Precondition Failed` (without calling its inner route) if the request comes with the respective conditional headers. Otherwise the request is simply passed on to its inner route.

The algorithm implemented by this directive closely follows what is defined in this section of the HTTPbis spec.

All responses (the ones produces by this directive itself as well as the ones coming back from the inner route) are augmented with respective `ETag` and `Last-Modified` response headers.

Since this directive requires the `EntityTag` and `lastModified` time stamp for the resource as concrete arguments it is usually used quite deep down in the route structure (i.e. close to the leaf-level), where the exact resource targeted by the request has already been established and the respective ETag/Last-Modified values can be determined.

The *FileAndResourceDirectives* internally use the `conditional` directive for ETag and Last-Modified support (if the `akka.http.routing.file-get-conditional` setting is enabled).

CodingDirectives

decodeRequest

Signature

```
def decodeRequest: Directive0
```

Description Decompresses the incoming request if it is `gzip` or `deflate` compressed. Uncompressed requests are passed through untouched. If the request encoded with another encoding the request is rejected with an `UnsupportedRequestEncodingRejection`.

Example

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

// tests:
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'"
}
```

decodeRequestWith

Signature

```
def decodeRequestWith(decoder: Decoder): Directive0
def decodeRequestWith(decoders: Decoder*): Directive0
```

Description Decodes the incoming request if it is encoded with one of the given encoders. If the request encoding doesn't match one of the given encoders the request is rejected with an `UnsupportedRequestEncodingRejection`. If no decoders are given the default encoders (`Gzip`, `Deflate`, `NoCoding`) are used.

Example

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

// tests:
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'")
    }
  }

// tests:
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'"
}
```

encodeResponse

Signature

```
def encodeResponse: Directive0
```

Description Encodes the response with the encoding that is requested by the client via the `Accept-Encoding` header or rejects the request with an `UnacceptedResponseEncodingRejection(supportedEncodings)`.

The response encoding is determined by the rules specified in RFC7231.

If the `Accept-Encoding` header is missing or empty or specifies an encoding other than identity, gzip or deflate then no encoding is used.

Example

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

// tests:
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)
}
```

encodeResponseWith

Signature

```
def encodeResponseWith(first: Encoder, more: Encoder*): Directive0
```

Description Encodes the response with the encoding that is requested by the client via the `Accept-Encoding` if it is among the provided encoders or rejects the request with an `UnacceptedResponseEncodingRejection(supportedEncodings)`.

The response encoding is determined by the rules specified in RFC7231.

If the `Accept-Encoding` header is missing then the response is encoded using the `first` encoder.

If the `Accept-Encoding` header is empty and `NoCoding` is part of the encoders then no response encoding is used. Otherwise the request is rejected.

Example

```
val route = encodeResponseWith(Gzip) { complete("content") }

// tests:
Get("/") ~> route ~> check {
  response should haveContentEncoding(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

Signature

```
def requestEncodedWith(encoding: HttpEncoding): Directive0
```

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

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

responseEncodingAccepted

Signature

```
def responseEncodingAccepted(encoding: HttpEncoding): Directive0
```

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

Example

```
val route = responseEncodingAccepted(gzip) { complete("content") }

Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "content"
}
Get("/") ~> `Accept-Encoding`(deflate) ~> route ~> check {
  rejection shouldEqual UnacceptedResponseEncodingRejection(gzip)
}
```

CookieDirectives

cookie

Signature

```
def cookie(name: String): Directive1[HttpCookiePair]
```

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

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.value}'")
  }

// tests:
Get("/") ~> Cookie("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

Signature

```
def deleteCookie(first: HttpCookie, more: HttpCookie*): Directive0
def deleteCookie(name: String, domain: String = "", path: String = ""): Directive0
```

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

Use the *setCookie* directive to update a cookie.

Example

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

// tests:
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "The user was logged out"
  header[`Set-Cookie`] shouldEqual Some(`Set-Cookie`(HttpCookie("userName", value = "deleted", ex
}
```

optionalCookie

Signature

```
def optionalCookie(name: String): Directive1[Option[HttpCookiePair]]
```

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

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.value}'")
    case None             => complete("No user logged in")
  }

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

setCookie

Signature

```
def setCookie(first: HttpCookie, more: HttpCookie*): Directive0
```

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

Use the *deleteCookie* directive to delete a cookie.

Example

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

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

DebuggingDirectives

logRequest

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. [3]

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
```

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

```
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))

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

logRequestResult

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. [4]

Description Logs both, the request and the response.

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

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

logResult

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

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. [5]

Description Logs the response.

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 any *RouteResult* coming back from the inner route.

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))

// 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))

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

ExecutionDirectives

handleExceptions

Signature

```
def handleExceptions(handler: ExceptionHandler): Directive0
```

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

Description Catches exceptions thrown by the inner route and handles them using the specified `ExceptionHandler`.

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 wr
}
val route =
  path("divide" / IntNumber / IntNumber) { (a, b) =>
    handleExceptions(divByZeroHandler) {
      complete(s"The result is ${a / b}")
    }
  }

// tests:
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

Signature

```
def handleRejections(handler: RejectionHandler): Directive0
```

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
  .handle { case ValidationRejection(msg, _) => complete((StatusCodes.InternalServerError, msg))
  .result()
val route =
  pathPrefix("handled") {
    handleRejections(totallyMissingHandler) {
      path("existing")(complete("This path exists")) ~
        path("boom")(reject(new ValidationRejection("This didn't work.")))
    }
  }

// tests:
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."
}
Get("/handled/boom") ~> route ~> check {
  status shouldEqual StatusCodes.InternalServerError
  responseAs[String] shouldEqual "This didn't work."
}
```

FileAndResourceDirectives Like the *RouteDirectives* the `FileAndResourceDirectives` are somewhat special in akka-http'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

Signature

```
def getFromBrowseableDirectories(directories: String*)(implicit renderer: DirectoryRenderer, reso
```

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.

For more details refer to *getFromBrowseableDirectory*.

Example

```
val route =
  path("tmp") {
    getFromBrowseableDirectories("/main", "/backups")
  }

// tests:
Get("/tmp") ~> route ~> check {
  status shouldEqual StatusCodes.OK
}
```

getFromBrowseableDirectory

Signature

```
def getFromBrowseableDirectory(directory: String)(implicit renderer: DirectoryRenderer, resolver:
```

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.

For more details refer to *getFromBrowseableDirectory*.

Example

```
val route =
  path("tmp") {
    getFromBrowseableDirectory("/tmp")
  }

// tests:
Get("/tmp") ~> route ~> check {
  status shouldEqual StatusCodes.OK
}
```

Default file listing page example Directives which list directories (e.g. `getFromBrowsableDirectory`) use an implicit `DirectoryRenderer` instance to perfm the actual rendering of the file listing. This rendered can be easily overriden by simply providing one in-scope for the directives to use, so you can build your custom directory listings.

The default renderer is `akka.http.scaladsl.server.directives.FileAndResourceDirectives.defaultDi` and renders a listing which looks like this:

Figure 2.1: Example page rendered by the `defaultDirectoryRenderer`.

It's possible to turn off rendering the footer stating which version of Akka HTTP is rendering this page by configuring the `akka.http.routing.render-vanity-footer` configuration option to `off`.

getFromDirectory

Signature

```
def getFromDirectory(directoryName: String)(implicit resolver: ContentTypeResolver): Route
```

Description Allows exposing a directory's files for GET requests for its contents.

The `unmatchedPath` (see *extractUnmatchedPath*) 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.

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.

Note: The file's contents will be read using an Akka Streams *Source* which *automatically uses a pre-configured dedicated blocking io dispatcher*, which separates the blocking file operations from the rest of the stream.

Note also that thanks to using Akka Streams internally, the file will be served at the highest speed reachable by the client, and not faster – i.e. the file will *not* end up being loaded in full into memory before writing it to the client.

Example

```
val route =
  path("tmp") {
    getFromDirectory("/tmp")
  }

// tests:
Get("/tmp/example") ~> route ~> check {
  responseAs[String] shouldEqual "example file contents"
}
```

getFromFile

Signature

```
def getFromFile(fileName: String)(implicit resolver: ContentTypeResolver): Route
def getFromFile(file: File)(implicit resolver: ContentTypeResolver): Route
def getFromFile(file: File, contentType: ContentType): Route
```

Description Allows exposing a file to be streamed to the client issuing the request.

The `unmatchedPath` (see *extractUnmatchedPath*) 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.

To files from a given directory use *getFromDirectory*. 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.

Note: The file's contents will be read using an Akka Streams *Source* which *automatically uses a pre-configured dedicated blocking io dispatcher*, which separates the blocking file operations from the rest of the stream.

Note also that thanks to using Akka Streams internally, the file will be served at the highest speed reachable by the client, and not faster – i.e. the file will *not* end up being loaded in full into memory before writing it to the client.

Example

```
import akka.http.scaladsl.server.directives._
import ContentTypeResolver.Default

val route =
  path("logs" / Segment) { name =>
    getFromFile(".log") // uses implicit ContentTypeResolver
  }

// tests:
Get("/logs/example") ~> route ~> check {
  responseAs[String] shouldEqual "example file contents"
}
```

getFromResource

Signature

```
def getFromResource(resourceName: String)(implicit resolver: ContentTypeResolver): Route
def getFromResource(resourceName: String, contentType: ContentType, classLoader: ClassLoader = de
```

Description Completes GET requests with the content of the given classpath resource.

For details refer to *getFromFile* which works the same way but obtaining the file from the filesystem instead of the applications classpath.

Note that it's not required to wrap this directive with `get` as this directive will only respond to `GET` requests.

Example

```
import akka.http.scaladsl.server.directives._
import ContentTypeResolver.Default

val route =
  path("logs" / Segment) { name =>
    getFromResource(".log") // uses implicit ContentTypeResolver
  }

// tests:
Get("/logs/example") ~> route ~> check {
  responseAs[String] shouldEqual "example file contents"
}
```

getFromResourceDirectory

Signature

```
def getFromResourceDirectory(directoryName: String, classLoader: ClassLoader = defaultClassLoader
```

Description Completes GET requests with the content of the given classpath resource directory.

For details refer to *getFromDirectory* which works the same way but obtaining the file from the filesystem instead of the applications classpath.

Note that it's not required to wrap this directive with `get` as this directive will only respond to `GET` requests.

Example

```
val route =
  path("examples") {
    getFromResourceDirectory("/examples")
  }

// tests:
Get("/examples/example-1") ~> route ~> check {
  responseAs[String] shouldEqual "example file contents"
}
```

listDirectoryContents

Signature

```
def listDirectoryContents(directories: String*)(implicit renderer: DirectoryRenderer): Route
```

Description 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]`.

To just serve files use *getFromDirectory*.

To serve files and provide a browseable directory listing use *getFromBrowseableDirectories* instead.

The rendering can be overridden by providing a custom `Marshaller[DirectoryListing]`, you can read more about it in *getFromDirectory* 's documentation.

Note that it's not required to wrap this directive with `get` as this directive will only respond to `GET` requests.

Example

```
val route =
  path("tmp") {
    listDirectoryContents("/tmp")
  } ~
    path("custom") {
      val renderer = new DirectoryRenderer {
        override def marshaller(renderVanityFooter: Boolean): ToEntityMarshaller[DirectoryListing
      }
      listDirectoryContents("/tmp")(renderer)
    }

// tests:
Get("/logs/example") ~> route ~> check {
  responseAs[String] shouldEqual "example file contents"
}
```

FileUploadDirectives

uploadedFile

Signature

```
def uploadedFile(fieldName: String): Directive1[(FileInfo, File)]
```

Description Streams the contents of a file uploaded as a multipart form into a temporary file on disk and provides the file and metadata about the upload as extracted value.

If there is an error writing to disk the request will be failed with the thrown exception, if there is no field with the given name the request will be rejected, if there are multiple file parts with the same name, the first one will be used and the subsequent ones ignored.

Note: This directive will stream contents of the request into a file, however one can not start processing these until the file has been written completely. For streaming APIs it is preferred to use the *fileUpload* directive, as it allows for streaming handling of the incoming data bytes.

Example

```
val route =
  uploadedFile("csv") {
    case (metadata, file) =>
      // do something with the file and file metadata ...
      file.delete()
      complete(StatusCodes.OK)
  }

// tests:
```

```
val multipartForm =
  Multipart.FormData(
    Multipart.FormData.BodyPart.Strict(
      "csv",
      HttpEntity(ContentTypes.`text/plain(UTF-8)`, "1,5,7\n11,13,17"),
      Map("filename" -> "data.csv")))

Post("/", multipartForm) ~> route ~> check {
  status shouldEqual StatusCodes.OK
}
```

fileUpload

Signature

```
def fileUpload(fieldName: String): Directive1[(FileInfo, Source[ByteString, Any])]
```

Description Simple access to the stream of bytes for a file uploaded as a multipart form together with metadata about the upload as extracted value.

If there is no field with the given name the request will be rejected, if there are multiple file parts with the same name, the first one will be used and the subsequent ones ignored.

Example

```
// adding integers as a service ;)
val route =
  extractRequestContext { ctx =>
    implicit val materializer = ctx.materializer
    implicit val ec = ctx.executionContext

    fileUpload("csv") {
      case (metadata, byteSource) =>

        val sumF: Future[Int] =
          // sum the numbers as they arrive so that we can
          // accept any size of file
          byteSource.via(Framing.delimiter(ByteString("\n"), 1024))
            .mapConcat(_.utf8String.split(",").toVector)
            .map(_.toInt)
            .runFold(0) { (acc, n) => acc + n }

        onSuccess(sumF) { sum => complete(s"Sum: $sum") }
    }
  }

// tests:
val multipartForm =
  Multipart.FormData(Multipart.FormData.BodyPart.Strict(
    "csv",
    HttpEntity(ContentTypes.`text/plain(UTF-8)`, "2,3,5\n7,11,13,17,23\n29,31,37\n"),
    Map("filename" -> "primes.csv")))

Post("/", multipartForm) ~> route ~> check {
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "Sum: 178"
}
```

```
curl --form "csv=@uploadFile.txt" http://<host>:<port>
```

FormFieldDirectives

formField

Signature

```
def formField(pdm: FieldMagnet): pdm.Out
```

Description Allows extracting a single Form field sent in the request.

See *formFields* for an in-depth description.

Example

```
val route =
  formFields('color, 'age.as[Int]) { (color, age) =>
    complete(s"The color is '$color' and the age ten years ago was ${age - 10}")
  }

// tests:
Post("/", FormData("color" -> "blue", "age" -> "68")) ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the age ten years ago was 58"
}

Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.BadRequest
  responseAs[String] shouldEqual "Request is missing required form field 'color'"
}
val route =
  formField('color) { color =>
    complete(s"The color is '$color'")
  } ~
    formField('id.as[Int]) { id =>
      complete(s"The id is '$id'")
    }

// tests:
Post("/", FormData("color" -> "blue")) ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue'"
}

Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.BadRequest
  responseAs[String] shouldEqual "Request is missing required form field 'color'"
}
```

formFields

Signature

```
def formFields(field: <FieldDef[T]>): Directive1[T]
def formFields(fields: <FieldDef[T_i]>*): Directive[T_0 :: ... T_i ... :: HNil]
def formFields(fields: <FieldDef[T_0]> :: ... <FieldDef[T_i]> ... :: HNil): Directive[T_0 :: ...
```

The signature shown is simplified and written in pseudo-syntax, the real signature uses magnets. [6] The type `<FieldDef>` doesn't really exist but consists of the syntactic variants as shown in the description and the examples.

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

Description Extracts fields from requests generated by HTML forms (independently of `HttpMethod` used).

Form fields can be either extracted as a String or can be converted to another type. The parameter name can be supplied either as a String or as a Symbol. Form field extraction can be modified to mark a field as required, optional, or repeated, or to filter requests where a form field has a certain value:

`"color"` extract value of field "color" as `String`

`"color".?` extract optional value of field "color" as `Option[String]`

`"color" ? "red"` extract optional value of field "color" as `String` with default value `"red"`

`"color" ! "blue"` require value of field "color" to be `"blue"` and extract nothing

`"amount".as[Int]` extract value of field "amount" as `Int`, you need a matching implicit `Unmarshaller` in scope for that to work (see also *Unmarshalling*)

`"amount".as(unmarshaller)` extract value of field "amount" with an explicit `Unmarshaller`

`"distance".*` extract multiple occurrences of field "distance" as `Iterable[String]`

`"distance".as[Int].*` extract multiple occurrences of field "distance" as `Iterable[Int]`, you need a matching implicit `Unmarshaller` in scope for that to work (see also *Unmarshalling*)

`"distance".as(unmarshaller).*` extract multiple occurrences of field "distance" with an explicit `Unmarshaller`

You can use *Case Class Extraction* to group several extracted values together into a case-class instance.

Requests missing a required field or field value will be rejected with an appropriate rejection.

There's also a singular version, *formField*.

Query parameters can be handled in a similar way, see *parameters*.

Unmarshalling Data POSTed from HTML forms is either of type `application/x-www-form-urlencoded` or of type `multipart/form-data`. The value of an url-encoded field is a `String` while the value of a `multipart/form-data`-encoded field is a "body part" containing an entity. This means that different kind of unmarshallers are needed depending on what the Content-Type of the request is:

- A `application/x-www-form-urlencoded` encoded field needs an implicit `Unmarshaller[Option[String], T]`
- A `multipart/form-data` encoded field needs an implicit `FromStrictFormFieldUnmarshaller[T]`

For common data-types, these implicits are predefined so that you usually don't need to care. For custom data-types it should usually suffice to create a `FromStringUnmarshaller[T]` if the value will be encoded as a `String`. This should be valid for all values generated by HTML forms apart from file uploads.

Details It should only be necessary to read and understand this paragraph if you have very special needs and need to process arbitrary forms, especially ones not generated by HTML forms.

The `formFields` directive contains this logic to find and decide how to deserialize a POSTed form field:

- It tries to find implicits of both types at the definition site if possible or otherwise at least one of both. If none is available compilation will fail with an "implicit not found" error.
- Depending on the `Content-Type` of the incoming request it first tries the matching (see above) one if available.
- If only a `Unmarshaller[Option[String], T]` is available when a request of type `multipart/form-data` is received, this unmarshaller will be tried to deserialize the body part for a field if the entity is of type `text/plain` or unspecified.
- If only a `FromStrictFormFieldUnmarshaller[T]` is available when a request of type `application/x-www-form-urlencoded` is received, this unmarshaller will be tried to deserialize

the field value by packing the field value into a body part with an entity of type `text/plain`. Deserializing will only succeed if the unmarshaller accepts entities of type `text/plain`.

If you need to handle encoded fields of a `multipart/form-data`-encoded request for a custom type, you therefore need to provide a `FromStrictFormFieldUnmarshaller[T]`.

Example

```
val route =
  formFields('color, 'age.as[Int]) { (color, age) =>
    complete(s"The color is '$color' and the age ten years ago was ${age - 10}")
  }

// tests:
Post("/", FormData("color" -> "blue", "age" -> "68")) ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the age ten years ago was 58"
}

Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.BadRequest
  responseAs[String] shouldEqual "Request is missing required form field 'color'"
}
```

For more examples about the way how fields can specified see the examples for the `parameters` directive.

FuturesDirectives Future directives can be used to run inner routes once the provided `Future[T]` has been completed.

onComplete

Signature

```
def onComplete[T](future: ⇒ Future[T]): Directive1[Try[T]]
```

Description Evaluates its parameter of type `Future[T]`, and once the `Future` has been completed, extracts its result as a value of type `Try[T]` and passes it to the inner route.

To handle the `Failure` case automatically and only work with the result value, use *onSuccess*.

To complete with a successful result automatically and just handle the failure result, use *completeOrRecoverWith*, instead.

Example

```
def divide(a: Int, b: Int): Future[Int] = Future {
  a / b
}

val route =
  path("divide" / IntNumber / IntNumber) { (a, b) =>
    onComplete(divide(a, b)) {
      case Success(value) => complete(s"The result was $value")
      case Failure(ex)    => complete((InternalServerError, s"An error occurred: ${ex.getMessage}"
    }
  }

// tests:
Get("/divide/10/2") ~> route ~> check {
  responseAs[String] shouldEqual "The result was 5"
}
```

```
Get("/divide/10/0") ~> Route.seal(route) ~> check {
  status shouldEqual InternalServerError
  responseAs[String] shouldEqual "An error occurred: / by zero"
}
```

onSuccess

Signature

```
def onSuccess(magnet: OnSuccessMagnet): Directive[magnet.Out]
```

Description Evaluates its parameter of type `Future[T]`, and once the `Future` has been completed successfully, extracts its result as a value of type `T` and passes it to the inner route.

If the future fails its failure throwable is bubbled up to the nearest `ExceptionHandler`.

To handle the `Failure` case manually as well, use *onComplete*, instead.

Example

```
val route =
  path("success") {
    onSuccess(Future { "Ok" }) { extraction =>
      complete(extraction)
    }
  } ~
  path("failure") {
    onSuccess(Future.failed[String](TestException)) { extraction =>
      complete(extraction)
    }
  }

// tests:
Get("/success") ~> route ~> check {
  responseAs[String] shouldEqual "Ok"
}

Get("/failure") ~> Route.seal(route) ~> check {
  status shouldEqual InternalServerError
  responseAs[String] shouldEqual "Unsuccessful future!"
}
```

completeOrRecoverWith

Signature

```
def completeOrRecoverWith(magnet: CompleteOrRecoverWithMagnet): Directive1[Throwable]
```

Description If the `Future[T]` succeeds the request is completed using the value's marshaller (this directive therefore requires a marshaller for the future's parameter type to be implicitly available). The execution of the inner route passed to this directive is only executed if the given future completed with a failure, exposing the reason of failure as a extraction of type `Throwable`.

To handle the successful case manually as well, use the *onComplete* directive, instead.

Example

```
val route =
  path("success") {
    completeOrRecoverWith(Future { "Ok" }) { extraction =>
      failWith(extraction) // not executed.
    }
  } ~
  path("failure") {
    completeOrRecoverWith(Future.failed[String](TestException)) { extraction =>
      failWith(extraction)
    }
  }

// tests:
Get("/success") ~> route ~> check {
  responseAs[String] shouldEqual "Ok"
}

Get("/failure") ~> Route.seal(route) ~> check {
  status shouldEqual InternalServerError
  responseAs[String] shouldEqual "Unsuccessful future!"
}
```

HeaderDirectives Header directives can be used to extract header values from the request. To change response headers use one of the *RespondWithDirectives*.

headerValue

Signature

```
def headerValue[T](f: HttpHeader ⇒ Option[T]): Directive1[T]
```

Description Traverses the list of request headers with the specified function and extracts the first value the function returns as `Some(value)`.

The *headerValue* directive is a mixture of `map` and `find` on the list of request headers. The specified function is called once for each header until the function returns `Some(value)`. This value is extracted and presented to the inner route. If the function throws an exception the request is rejected with a `MalformedHeaderRejection`. If the function returns `None` for every header the request is rejected as "NotFound".

This directive is the basis for building other request header related directives.

See also *headerValuePF* for a nicer syntactic alternative.

Example

```
def extractHostPort: HttpHeader => Option[Int] = {
  case h: `Host` => Some(h.port)
  case x         => None
}

val route =
  headerValue(extractHostPort) { port =>
    complete(s"The port was $port")
  }

// tests:
Get("/") ~> Host("example.com", 5043) ~> route ~> check {
  responseAs[String] shouldEqual "The port was 5043"
}
Get("/") ~> Route.seal(route) ~> check {
```

```
  status shouldEqual NotFound
  responseAs[String] shouldEqual "The requested resource could not be found."
}
```

headerValueByName

Signature

```
def headerValueByName(headerName: Symbol): Directive1[String]
def headerValueByName(headerName: String): Directive1[String]
```

Description Extracts the value of the HTTP request header with the given name.

The name can be given as a `String` or as a `Symbol`. If no header with a matching name is found the request is rejected with a `MissingHeaderRejection`.

If the header is expected to be missing in some cases or to customize handling when the header is missing use the *optionalHeaderValueByName* directive instead.

Example

```
val route =
  headerValueByName("X-User-Id") { userId =>
    complete(s"The user is $userId")
  }

// tests:
Get("/") ~> RawHeader("X-User-Id", "Joe42") ~> route ~> check {
  responseAs[String] shouldEqual "The user is Joe42"
}

Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual BadRequest
  responseAs[String] shouldEqual "Request is missing required HTTP header 'X-User-Id'"
}
```

headerValueByType

Signature

```
def headerValueByType[T <: HttpHeader: ClassTag](): Directive1[T]
```

The signature shown is simplified, the real signature uses magnets. [7]

Description Traverses the list of request headers and extracts the first header of the given type.

The `headerValueByType` directive finds a header of the given type in the list of request header. If no header of the given type is found the request is rejected with a `MissingHeaderRejection`.

If the header is expected to be missing in some cases or to customize handling when the header is missing use the *optionalHeaderValueByType* directive instead.

Example

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

```
val route =
  headerValueByType[Origin]() { origin ⇒
    complete(s"The first origin was ${origin.origins.head}")
  }

val originHeader = Origin(HttpOrigin("http://localhost:8080"))

// tests:
// extract a header if the type is matching
Get("abc") ~> originHeader ~> route ~> check {
  responseAs[String] shouldEqual "The first origin was http://localhost:8080"
}

// reject a request if no header of the given type is present
Get("abc") ~> route ~> check {
  inside(rejection) { case MissingHeaderRejection("Origin") ⇒ }
}
```

headerValuePF

Signature

```
def headerValuePF[T](pf: PartialFunction[HttpHeader, T]): Directive1[T]
```

Description Calls the specified partial function with the first request header the function is `isDefinedAt` and extracts the result of calling the function.

The `headerValuePF` directive is an alternative syntax version of *headerValue*.

If the function throws an exception the request is rejected with a `MalformedHeaderRejection`.

If the function is not defined for any header the request is rejected as "NotFound".

Example

```
def extractHostPort: PartialFunction[HttpHeader, Int] = {
  case h: `Host` => h.port
}

val route =
  headerValuePF(extractHostPort) { port =>
    complete(s"The port was $port")
  }

// tests:
Get("/") ~> Host("example.com", 5043) ~> route ~> check {
  responseAs[String] shouldEqual "The port was 5043"
}
Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual NotFound
  responseAs[String] shouldEqual "The requested resource could not be found."
}
```

optionalHeaderValue

Signature

```
def optionalHeaderValue[T](f: HttpHeader ⇒ Option[T]): Directive1[Option[T]]
```

Description Traverses the list of request headers with the specified function and extracts the first value the function returns as `Some(value)`.

The `optionalHeaderValue` directive is similar to the *headerValue* directive but always extracts an `Option` value instead of rejecting the request if no matching header could be found.

Example

```
def extractHostPort: HttpHeader => Option[Int] = {
  case h: `Host` => Some(h.port)
  case x         => None
}

val route =
  optionalHeaderValue(extractHostPort) {
    case Some(port) => complete(s"The port was $port")
    case None       => complete(s"The port was not provided explicitly")
  } ~ // can also be written as:
    optionalHeaderValue(extractHostPort) { port =>
      complete {
        port match {
          case Some(p) => s"The port was $p"
          case _       => "The port was not provided explicitly"
        }
      }
    }

// tests:
Get("/") ~> Host("example.com", 5043) ~> route ~> check {
  responseAs[String] shouldEqual "The port was 5043"
}
Get("/") ~> Route.seal(route) ~> check {
  responseAs[String] shouldEqual "The port was not provided explicitly"
}
```

optionalHeaderValueByName

Signature

```
def optionalHeaderValueByName(headerName: Symbol): Directive1[Option[String]]
def optionalHeaderValueByName(headerName: String): Directive1[Option[String]]
```

Description Optionally extracts the value of the HTTP request header with the given name.

The `optionalHeaderValueByName` directive is similar to the *headerValueByName* directive but always extracts an `Option` value instead of rejecting the request if no matching header could be found.

Example

```
val route =
  optionalHeaderValueByName("X-User-Id") {
    case Some(userId) => complete(s"The user is $userId")
    case None         => complete(s"No user was provided")
  } ~ // can also be written as:
    optionalHeaderValueByName("port") { port =>
      complete {
        port match {
          case Some(p) => s"The user is $p"
          case _       => "No user was provided"
        }
      }
```

```
    }

// tests:
Get("/") ~> RawHeader("X-User-Id", "Joe42") ~> route ~> check {
  responseAs[String] shouldEqual "The user is Joe42"
}
Get("/") ~> Route.seal(route) ~> check {
  responseAs[String] shouldEqual "No user was provided"
}
```

optionalHeaderValueByType

Signature

```
def optionalHeaderValueByType[T <: HttpHeader: ClassTag](): Directive1[Option[T]]
```

The signature shown is simplified, the real signature uses magnets. [8]

Description Optionally extracts the value of the HTTP request header of the given type.

The `optionalHeaderValueByType` directive is similar to the *headerValueByType* directive but always extracts an `Option` value instead of rejecting the request if no matching header could be found.

Example

```
val route =
  optionalHeaderValueByType[Origin]() {
    case Some(origin) ⇒ complete(s"The first origin was ${origin.origins.head}")
    case None         ⇒ complete("No Origin header found.")
  }

val originHeader = Origin(HttpOrigin("http://localhost:8080"))

// tests:
// extract Some(header) if the type is matching
Get("abc") ~> originHeader ~> route ~> check {
  responseAs[String] shouldEqual "The first origin was http://localhost:8080"
}

// extract None if no header of the given type is present
Get("abc") ~> route ~> check {
  responseAs[String] shouldEqual "No Origin header found."
}
```

optionalHeaderValuePF

Signature

```
def optionalHeaderValuePF[T](pf: PartialFunction[HttpHeader, T]): Directive1[Option[T]]
```

Description Calls the specified partial function with the first request header the function is `isDefinedAt` and extracts the result of calling the function.

The `optionalHeaderValuePF` directive is similar to the *headerValuePF* directive but always extracts an `Option` value instead of rejecting the request if no matching header could be found.

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

Example

```
def extractHostPort: PartialFunction[HttpHeader, Int] = {
  case h: `Host` => h.port
}

val route =
  optionalHeaderValuePF(extractHostPort) {
    case Some(port) => complete(s"The port was $port")
    case None       => complete(s"The port was not provided explicitly")
  } ~ // can also be written as:
    optionalHeaderValuePF(extractHostPort) { port =>
      complete {
        port match {
          case Some(p) => s"The port was $p"
          case _       => "The port was not provided explicitly"
        }
      }
    }

// tests:
Get("/") ~> Host("example.com", 5043) ~> route ~> check {
  responseAs[String] shouldEqual "The port was 5043"
}
Get("/") ~> Route.seal(route) ~> check {
  responseAs[String] shouldEqual "The port was not provided explicitly"
}
```

HostDirectives HostDirectives allow you to filter requests based on the hostname part of the `Host` header contained in incoming requests as well as extracting its value for usage in inner routes.

extractHost

Signature

```
def extractHost: Directive1[String]
```

Description Extract the hostname part of the `Host` request header and expose it as a `String` extraction to its inner route.

Example

```
val route =
  extractHost { hn =>
    complete(s"Hostname: $hn")
  }

// tests:
Get() ~> Host("company.com", 9090) ~> route ~> check {
  status shouldEqual OK
  responseAs[String] shouldEqual "Hostname: company.com"
}
```

host

Signature

```
def host(hostNames: String*): Directive0
/**
 * Rejects all requests for whose host name the given predicate function returns false.
 */
def host(predicate: String ⇒ Boolean): Directive0 = extractHost.require(predicate)
def host(regex: Regex): Directive1[String]
```

Description Filter requests matching conditions against the hostname part of the Host header value in the request.

The `def host(hostNames: String*)` overload rejects all requests with a hostname different from the given ones.

The `def host(predicate: String ⇒ Boolean)` overload rejects all requests for which the hostname does not satisfy the given predicate.

The `def host(regex: Regex)` overload works a little bit different: it rejects all requests with a hostname that doesn't have a prefix matching the given regular expression and also extracts a `String` to its inner route following this rules:

- For all matching requests the prefix string matching the regex is extracted and passed to the inner route.
- If the regex contains a capturing group only the string matched by this group is extracted.
- If the regex contains more than one capturing group an `IllegalArgumentException` is thrown.

Example Matching a list of hosts:

```
val route =
  host("api.company.com", "rest.company.com") {
    complete("Ok")
  }

// tests:
Get() ~> Host("rest.company.com") ~> route ~> check {
  status shouldEqual OK
  responseAs[String] shouldEqual "Ok"
}

Get() ~> Host("notallowed.company.com") ~> route ~> check {
  handled shouldBe false
}
```

Making sure the host satisfies the given predicate

```
val shortOnly: String => Boolean = (hostname) => hostname.length < 10

val route =
  host(shortOnly) {
    complete("Ok")
  }

// tests:
Get() ~> Host("short.com") ~> route ~> check {
  status shouldEqual OK
  responseAs[String] shouldEqual "Ok"
}

Get() ~> Host("verylonghostname.com") ~> route ~> check {
  handled shouldBe false
}
```

Using a regular expressions:

```
val route =
  host("api|rest".r) { prefix =>
    complete(s"Extracted prefix: $prefix")
  } ~
    host("public.(my|your)company.com".r) { captured =>
      complete(s"You came through $captured company")
    }

// tests:
Get() ~> Host("api.company.com") ~> route ~> check {
  status shouldEqual OK
  responseAs[String] shouldEqual "Extracted prefix: api"
}

Get() ~> Host("public.mycompany.com") ~> route ~> check {
  status shouldEqual OK
  responseAs[String] shouldEqual "You came through my company"
}
```

Beware that in the case of introducing multiple capturing groups in the regex such as in the case bellow, the directive will fail at runtime, at the moment the route tree is evaluated for the first time. This might cause your http handler actor to enter in a fail/restart loop depending on your supervision strategy.

```
an[IllegalArgumentException] should be thrownBy {
  host("server-([0-9]).company.(com|net|org)".r) { target =>
    complete("Will never complete :'(")
  }
}
```

Marshalling Directives Marshalling directives work in conjunction with `akka.http.scaladsl.marshalling` and `akka.http.scaladsl.unmarshalling` to convert a request entity to a specific type or a type to a response.

See *marshalling* and *unmarshalling* for specific serialization (also known as pickling) guidance.

Marshalling directives usually rely on an in-scope implicit marshaller to handle conversion.

completeWith

Signature

```
def completeWith[T](marshaller: ToResponseMarshaller[T])(inner: (T ⇒ Unit) ⇒ Unit): Route
```

Description Uses the marshaller for a given type to produce a completion function that is passed to its inner route. You can use it to decouple marshaller resolution from request completion.

The `completeWith` directive works in conjuction with `instanceOf` and `spray.httpx.marshalling` to convert higher-level (object) structure into some lower-level serialized "wire format". *The marshalling documentation* explains this process in detail. This directive simplifies exposing types to clients via a route while providing some form of access to the current context.

`completeWith` is similar to `handleWith`. The main difference is with `completeWith` you must eventually call the completion function generated by `completeWith`. `handleWith` will automatically call `complete` when the `handleWith` function returns.

Examples The following example uses `spray-json` to marshall a simple `Person` class to a json response. It utilizes `SprayJsonSupport` via the `PersonJsonSupport` object as the in-scope unmarshaller.

```
object PersonJsonSupport extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val PortofolioFormats = jsonFormat2(Person)
}
```

```
case class Person(name: String, favoriteNumber: Int)
```

The `findPerson` takes an argument of type `Person => Unit` which is generated by the `completeWith` call. We can handle any logic we want in `findPerson` and call our completion function to complete the request.

```
import PersonJsonSupport._

val findPerson = (f: Person => Unit) => {

  //... some processing logic...

  //complete the request
  f(Person("Jane", 42))
}

val route = get {
  completeWith(instanceOf[Person]) { completionFunction => findPerson(completionFunction) }
}

// tests:
Get("/") ~> route ~> check {
  mediaType shouldEqual `application/json`
  responseAs[String] should include("""name": "Jane"""")
  responseAs[String] should include("""favoriteNumber": 42""")
}
```

entity

Signature

```
def entity[T](um: FromRequestUnmarshaller[T]): Directive1[T]
```

Description Unmarshalls the request entity to the given type and passes it to its inner Route. An unmarshaller returns an `Either` with `Right(value)` if successful or `Left(exception)` for a failure. The `entity` method will either pass the `value` to the inner route or map the `exception` to a `akka.http.scaladsl.server.Rejection`.

The `entity` directive works in conjuction with `as` and `akka.http.scaladsl.unmarshalling` to convert some serialized "wire format" value into a higher-level object structure. *The unmarshalling documentation* explains this process in detail. This directive simplifies extraction and error handling to the specified type from the request.

An unmarshaller will return a `Left(exception)` in the case of an error. This is converted to a `akka.http.scaladsl.server.Rejection` within the `entity` directive. The following table lists how exceptions are mapped to rejections:

Left(exception)	Rejection
`ContentExpected`	`RequestEntityExpectedRejection`
`UnsupportedContentType`	`UnsupportedRequestContentTypeRejection`, which lists the supported types
`MaformedContent`	`MalformedRequestContentRejection`, with an error message and cause

Examples The following example uses `spray-json` to unmarshall a json request into a simple `Person` class. It utilizes `SprayJsonSupport` via the `PersonJsonSupport` object as the in-scope unmarshaller.

```
case class Person(name: String, favoriteNumber: Int)
```

```
object PersonJsonSupport extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val PortofolioFormats = jsonFormat2(Person)
}
```

```
import PersonJsonSupport._

val route = post {
  entity(as[Person]) { person =>
    complete(s"Person: ${person.name} - favorite number: ${person.favoriteNumber}")
  }
}

// tests:
Post("/", HttpEntity(`application/json`, """{ "name": "Jane", "favoriteNumber" : 42 }""")) ~>
  route ~> check {
    responseAs[String] shouldEqual "Person: Jane - favorite number: 42"
  }
```

handleWith

Signature

```
def handleWith[A, B](f: A ⇒ B)(implicit um: FromRequestUnmarshaller[A], m: ToResponseMarshaller[B
```

Description Completes the request using the given function. The input to the function is produced with the in-scope entity unmarshaller and the result value of the function is marshalled with the in-scope marshaller. `handleWith` can be a convenient method combining `entity` with `complete`.

The `handleWith` directive is used when you want to handle a route with a given function of type A ⇒ B. `handleWith` will use both an in-scope unmarshaller to convert a request into type A and an in-scope marshaller to convert type B into a response. This is helpful when your core business logic resides in some other class or you want your business logic to be independent of the REST interface written with akka-http. You can use `handleWith` to "hand off" processing to a given function without requiring any akka-http-specific functionality.

`handleWith` is similar to `produce`. The main difference is `handleWith` automatically calls `complete` when the function passed to `handleWith` returns. Using `produce` you must explicity call the completion function passed from the `produce` function.

See *marshalling* and *unmarshalling* for guidance on marshalling entities with akka-http.

Examples The following example uses an `updatePerson` function with a `Person` case class as an input and output. We plug this function into our route using `handleWith`.

```
case class Person(name: String, favoriteNumber: Int)
```

```
import PersonJsonSupport._

val updatePerson = (person: Person) => {

  //... some processing logic...

  //return the person
  person
}

val route = post {
  handleWith(updatePerson)
}
```

```
// tests:
Post("/", HttpEntity(`application/json`, """{ "name": "Jane", "favoriteNumber" : 42 }""")) ~>
  route ~> check {
    mediaType shouldEqual `application/json`
    responseAs[String] should include("""name": "Jane"""")
    responseAs[String] should include("""favoriteNumber": 42""")
  }
```

The PersonJsonSupport object handles both marshalling and unmarshalling of the Person case class.

```
object PersonJsonSupport extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val PortofolioFormats = jsonFormat2(Person)
}
```

Understanding Specific Marshalling Directives

directive	behavior
completeWith	Uses a marshaller for a given type to produce a completion functi conjuction with *instanceOf* to format responses.
entity	Unmarshalls the request entity to the given type and passes it to it conjection with *as* to convert requests to objects.
handleWith	Completes a request with a given function, using an in-scope unm in-scope marshaller for the output.

MethodDirectives

delete

Signature

```
def delete: Directive0
```

Description Matches requests with HTTP method `DELETE`.

This directive filters an incoming request by its HTTP method. Only requests with method `DELETE` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a `405 Method Not Allowed` response by the default *RejectionHandler*.

Example

```
val route = delete { complete("This is a DELETE request.") }

// tests:
Delete("/") ~> route ~> check {
  responseAs[String] shouldEqual "This is a DELETE request."
}
```

extractMethod

Signature

```
def extractMethod: Directive1[HttpMethod]
```

Description Extracts the `HttpMethod` from the request context and provides it for use for other directives explicitly.

Example In the below example our route first matches all `GET` requests, and if an incoming request wasn't a `GET`, the matching continues and the extractMethod route will be applied which we can use to programatically print what type of request it was - independent of what actual HttpMethod it was:

```
val route =
  get {
    complete("This is a GET request.")
  } ~
    extractMethod { method =>
      complete(s"This ${method.name} request, clearly is not a GET!")
    }

// tests:
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "This is a GET request."
}

Put("/") ~> route ~> check {
  responseAs[String] shouldEqual "This PUT request, clearly is not a GET!"
}
Head("/") ~> route ~> check {
  responseAs[String] shouldEqual "This HEAD request, clearly is not a GET!"
}
```

get

Signature

```
def get: Directive0
```

Description Matches requests with HTTP method `GET`.

This directive filters the incoming request by its HTTP method. Only requests with method `GET` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a `405 Method Not Allowed` response by the default *RejectionHandler*.

Example

```
val route = get { complete("This is a GET request.") }

// tests:
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "This is a GET request."
}
```

head

Signature

```
def head: Directive0
```

Description Matches requests with HTTP method `HEAD`.

This directive filters the incoming request by its HTTP method. Only requests with method `HEAD` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a `405 Method Not Allowed` response by the default *RejectionHandler*.

Note: By default, akka-http handles HEAD-requests transparently by dispatching a GET-request to the handler

and stripping of the result body. See the `akka.http.server.transparent-head-requests` setting for how to disable this behavior.

Example

```
val route = head { complete("This is a HEAD request.") }

// tests:
Head("/") ~> route ~> check {
  responseAs[String] shouldEqual "This is a HEAD request."
}
```

method

Signature

```
/**
 * Rejects all requests whose HTTP method does not match the given one.
 */
def method(httpMethod: HttpMethod): Directive0 =
  extractMethod.flatMap[Unit] {
    case `httpMethod` ⇒ pass
    case _            ⇒ reject(MethodRejection(httpMethod))
  } & cancelRejections(classOf[MethodRejection])
```

Description Matches HTTP requests based on their method.

This directive filters the incoming request by its HTTP method. Only requests with the specified method are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a `405 Method Not Allowed` response by the default *RejectionHandler*.

Example

```
val route = method(HttpMethods.PUT) { complete("This is a PUT request.") }

// tests:
Put("/", "put content") ~> route ~> check {
  responseAs[String] shouldEqual "This is a PUT request."
}

Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.MethodNotAllowed
  responseAs[String] shouldEqual "HTTP method not allowed, supported methods: PUT"
}
```

options

Signature

```
def options: Directive0
```

Description Matches requests with HTTP method `OPTIONS`.

This directive filters the incoming request by its HTTP method. Only requests with method `OPTIONS` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a `405 Method Not Allowed` response by the default *RejectionHandler*.

Example

```
val route = options { complete("This is an OPTIONS request.") }

// tests:
Options("/") ~> route ~> check {
  responseAs[String] shouldEqual "This is an OPTIONS request."
}
```

overrideMethodWithParameter

Signature

```
def overrideMethodWithParameter(paramName: String): Directive0
```

Description Changes the request method to the value of the specified query parameter.

Changes the HTTP method of the request to the value of the specified query string parameter. If the query string parameter is not specified this directive has no effect.

If the query string is specified as something that is not a HTTP method, then this directive completes the request with a `501 Not Implemented` response.

This directive is useful for:

- Use in combination with JSONP (JSONP only supports GET)
- Supporting older browsers that lack support for certain HTTP methods. E.g. IE8 does not support PATCH

Example

```
val route =
  overrideMethodWithParameter("method") {
    get {
      complete("This looks like a GET request.")
    } ~
      post {
        complete("This looks like a POST request.")
      }
  }

// tests:
Get("/?method=POST") ~> route ~> check {
  responseAs[String] shouldEqual "This looks like a POST request."
}
Post("/?method=get") ~> route ~> check {
  responseAs[String] shouldEqual "This looks like a GET request."
}

Get("/?method=hallo") ~> route ~> check {
  status shouldEqual StatusCodes.NotImplemented
}
```

patch

Signature

```
def patch: Directive0
```

Description Matches requests with HTTP method `PATCH`.

This directive filters the incoming request by its HTTP method. Only requests with method `PATCH` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a `405 Method Not Allowed` response by the default *RejectionHandler*.

Example

```
val route = patch { complete("This is a PATCH request.") }

// tests:
Patch("/", "patch content") ~> route ~> check {
  responseAs[String] shouldEqual "This is a PATCH request."
}
```

post

Signature

```
def post: Directive0
```

Description Matches requests with HTTP method `POST`.

This directive filters the incoming request by its HTTP method. Only requests with method `POST` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a `405 Method Not Allowed` response by the default *RejectionHandler*.

Example

```
val route = post { complete("This is a POST request.") }

// tests:
Post("/", "post content") ~> route ~> check {
  responseAs[String] shouldEqual "This is a POST request."
}
```

put

Signature

```
def put: Directive0
```

Description Matches requests with HTTP method `PUT`.

This directive filters the incoming request by its HTTP method. Only requests with method `PUT` are passed on to the inner route. All others are rejected with a `MethodRejection`, which is translated into a `405 Method Not Allowed` response by the default *RejectionHandler*.

Example

```
val route = put { complete("This is a PUT request.") }

// tests:
Put("/", "put content") ~> route ~> check {
  responseAs[String] shouldEqual "This is a PUT request."
}
```

MiscDirectives

extractClientIP

Signature

```
def extractClientIP: Directive1[RemoteAddress]
```

Description Provides the value of `X-Forwarded-For`, `Remote-Address`, or `X-Real-IP` headers as an instance of `HttpIp`.

The akka-http server engine adds the `Remote-Address` header to every request automatically if the respective setting `akka.http.server.remote-address-header` is set to `on`. Per default it is set to `off`.

Example

```
val route = extractClientIP { ip =>
  complete("Client's ip is " + ip.toOption.map(_.getHostAddress).getOrElse("unknown"))
}

// tests:
Get("/").withHeaders(`Remote-Address`(RemoteAddress("192.168.3.12"))) ~> route ~> check {
  responseAs[String] shouldEqual "Client's ip is 192.168.3.12"
}
```

rejectEmptyResponse

Signature

```
def rejectEmptyResponse: Directive0
```

Description Replaces a response with no content with an empty rejection.

The `rejectEmptyResponse` directive is mostly used with marshalling `Option[T]` instances. The value `None` is usually marshalled to an empty but successful result. In many cases `None` should instead be handled as `404 Not Found` which is the effect of using `rejectEmptyResponse`.

Example

```
val route = rejectEmptyResponse {
  path("even" / IntNumber) { i =>
    complete {
      // returns Some(evenNumberDescription) or None
      Option(i).filter(_ % 2 == 0).map { num =>
        s"Number $num is even."
      }
    }
  }
}

// tests:
Get("/even/23") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.NotFound
}
Get("/even/28") ~> route ~> check {
  responseAs[String] shouldEqual "Number 28 is even."
}
```

requestEntityEmpty

Signature

```
def requestEntityEmpty: Directive0
```

Description A filter that checks if the request entity is empty and only then passes processing to the inner route. Otherwise, the request is rejected.

See also *requestEntityPresent* for the opposite effect.

Example

```
val route =
  requestEntityEmpty {
    complete("request entity empty")
  } ~
    requestEntityPresent {
      complete("request entity present")
    }

// tests:
Post("/", "text") ~> Route.seal(route) ~> check {
  responseAs[String] shouldEqual "request entity present"
}
Post("/") ~> route ~> check {
  responseAs[String] shouldEqual "request entity empty"
}
```

requestEntityPresent

Signature

```
def requestEntityPresent: Directive0
```

Description A simple filter that checks if the request entity is present and only then passes processing to the inner route. Otherwise, the request is rejected.

See also *requestEntityEmpty* for the opposite effect.

Example

```
val route =
  requestEntityEmpty {
    complete("request entity empty")
  } ~
    requestEntityPresent {
      complete("request entity present")
    }

// tests:
Post("/", "text") ~> Route.seal(route) ~> check {
  responseAs[String] shouldEqual "request entity present"
}
Post("/") ~> route ~> check {
  responseAs[String] shouldEqual "request entity empty"
}
```

selectPreferredLanguage

Signature

```
def selectPreferredLanguage(first: Language, more: Language*): Directive1[Language]
```

Description Inspects the request's `Accept-Language` header and determines, which of a given set of language alternatives is preferred by the client according to content negotiation rules defined by http://tools.ietf.org/html/rfc7231#section-5.3.5.

If there are several best language alternatives that the client has equal preference for (even if this preference is zero!) the order of the arguments is used as a tie breaker (first one wins).

Example

```
val request = Get() ~> `Accept-Language`(
  Language("en-US"),
  Language("en") withQValue 0.7f,
  LanguageRange.`*` withQValue 0.1f,
  Language("de") withQValue 0.5f)

request ~> {
  selectPreferredLanguage("en", "en-US") { lang ⇒
    complete(lang.toString)
  }
} ~> check { responseAs[String] shouldEqual "en-US" }

request ~> {
  selectPreferredLanguage("de-DE", "hu") { lang ⇒
    complete(lang.toString)
  }
} ~> check { responseAs[String] shouldEqual "de-DE" }
```

validate

Signature

```
def validate(check: ⇒ Boolean, errorMsg: String): Directive0
```

Description Allows validating a precondition before handling a route.

Checks an arbitrary condition and passes control to the inner route if it returns `true`. Otherwise, rejects the request with a `ValidationRejection` containing the given error message.

Example

```
val route =
  extractUri { uri =>
    validate(uri.path.toString.size < 5, s"Path too long: '${uri.path.toString}'") {
      complete(s"Full URI: $uri")
    }
  }

// tests:
Get("/234") ~> route ~> check {
  responseAs[String] shouldEqual "Full URI: http://example.com/234"
}
Get("/abcdefghijkl") ~> route ~> check {
  rejection shouldEqual ValidationRejection("Path too long: '/abcdefghijkl'", None)
}
```

ParameterDirectives

parameter

Signature

```
def parameter(pdm: ParamMagnet): pdm.Out
```

Description Extracts a *query* parameter value from the request.

See *parameters* for a detailed description of this directive.

See *When to use which parameter directive?* to understand when to use which directive.

Example

```
val route =
  parameter('color) { color =>
    complete(s"The color is '$color'")
  }

// tests:
Get("/?color=blue") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue'"
}

Get("/") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.NotFound
  responseAs[String] shouldEqual "Request is missing required query parameter 'color'"
}
```

parameterMap

Signature

```
def parameterMap: Directive1[Map[String, String]]
```

Description Extracts all parameters at once as a `Map[String, String]` mapping parameter names to parameter values.

If a query contains a parameter value several times, the map will contain the last one.

See also *When to use which parameter directive?* to understand when to use which directive.

Example

```
val route =
  parameterMap { params =>
    def paramString(param: (String, String)): String = s"""${param._1} = '${param._2}'"""
    complete(s"The parameters are ${params.map(paramString).mkString(", ")}")
  }

// tests:
Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are color = 'blue', count = '42'"
}
Get("/?x=1&x=2") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are x = '2'"
}
```

parameterMultiMap

Signature

```
def parameterMultiMap: Directive1[Map[String, List[String]]]
```

Description Extracts all parameters at once as a multi-map of type `Map[String, List[String]]` mapping a parameter name to a list of all its values.

This directive can be used if parameters can occur several times.

The order of values is *not* specified.

See *When to use which parameter directive?* to understand when to use which directive.

Example

```
val route =
  parameterMultiMap { params =>
    complete(s"There are parameters ${params.map(x => x._1 + " -> " + x._2.size).mkString(", ")}"
  }

// tests:
Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] shouldEqual "There are parameters color -> 1, count -> 1"
}
Get("/?x=23&x=42") ~> route ~> check {
  responseAs[String] shouldEqual "There are parameters x -> 2"
}
```

parameters

Signature

```
def parameters(param: <ParamDef[T]>): Directive1[T]
def parameters(params: <ParamDef[T_i]>*): Directive[T_0 :: ... T_i ... :: HNil]
def parameters(params: <ParamDef[T_0]> :: ... <ParamDef[T_i]> ... :: HNil): Directive[T_0 :: ...
```

The signature shown is simplified and written in pseudo-syntax, the real signature uses magnets. [9] The type `<ParamDef>` doesn't really exist but consists of the syntactic variants as shown in the description and the examples.

Description The parameters directive filters on the existence of several query parameters and extract their values.

Query parameters can be either extracted as a String or can be converted to another type. The parameter name can be supplied either as a String or as a Symbol. Parameter extraction can be modified to mark a query parameter as required, optional, or repeated, or to filter requests where a parameter has a certain value:

`"color"` extract value of parameter "color" as `String`

`"color".?` extract optional value of parameter "color" as `Option[String]`

`"color" ? "red"` extract optional value of parameter "color" as `String` with default value `"red"`

`"color" ! "blue"` require value of parameter "color" to be `"blue"` and extract nothing

`"amount".as[Int]` extract value of parameter "amount" as `Int`, you need a matching `Deserializer` in scope for that to work (see also *Unmarshalling*)

`"amount".as(deserializer)` extract value of parameter "amount" with an explicit `Deserializer`

`"distance".*` extract multiple occurrences of parameter "distance" as `Iterable[String]`

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

`"distance".as[Int].*` extract multiple occurrences of parameter "distance" as `Iterable[Int]`, you need a matching `Deserializer` in scope for that to work (see also *Unmarshalling*)

`"distance".as(deserializer).*` extract multiple occurrences of parameter "distance" with an explicit `Deserializer`

You can use *Case Class Extraction* to group several extracted values together into a case-class instance.

Requests missing a required parameter or parameter value will be rejected with an appropriate rejection.

There's also a singular version, *parameter*. Form fields can be handled in a similar way, see `formFields`. If you want unified handling for both query parameters and form fields, see `anyParams`.

Examples

Required parameter

```
val route =
  parameters('color, 'backgroundColor) { (color, backgroundColor) =>
    complete(s"The color is '$color' and the background is '$backgroundColor'")
  }

// tests:
Get("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'red'"
}
Get("/?color=blue") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.NotFound
  responseAs[String] shouldEqual "Request is missing required query parameter 'backgroundColor'"
}
```

Optional parameter

```
val route =
  parameters('color, 'backgroundColor.?) { (color, backgroundColor) =>
    val backgroundStr = backgroundColor.getOrElse("<undefined>")
    complete(s"The color is '$color' and the background is '$backgroundStr'")
  }

// tests:
Get("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'red'"
}
Get("/?color=blue") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is '<undefined>'"
}
val route =
  parameters('color, 'backgroundColor ? "white") { (color, backgroundColor) =>
    complete(s"The color is '$color' and the background is '$backgroundColor'")
  }

// tests:
Get("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'red'"
}
Get("/?color=blue") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'white'"
}
```

Optional parameter with default value

```
val route =
  parameters('color, 'backgroundColor ? "white") { (color, backgroundColor) =>
    complete(s"The color is '$color' and the background is '$backgroundColor'")
  }

// tests:
Get("/?color=blue&backgroundColor=red") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'red'"
}
Get("/?color=blue") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and the background is 'white'"
}
```

Parameter with required value

```
val route =
  parameters('color, 'action ! "true") { (color) =>
    complete(s"The color is '$color'.")
  }

// tests:
Get("/?color=blue&action=true") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue'."
}

Get("/?color=blue&action=false") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.NotFound
  responseAs[String] shouldEqual "The requested resource could not be found."
}
```

Deserialized parameter

```
val route =
  parameters('color, 'count.as[Int]) { (color, count) =>
    complete(s"The color is '$color' and you have $count of it.")
  }

// tests:
Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] shouldEqual "The color is 'blue' and you have 42 of it."
}

Get("/?color=blue&count=blub") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.BadRequest
  responseAs[String] shouldEqual "The query parameter 'count' was malformed:\n'blub' is not a vali
}
```

Repeated parameter

```
val route =
  parameters('color, 'city.*) { (color, cities) =>
    cities.toList match {
      case Nil         => complete(s"The color is '$color' and there are no cities.")
      case city :: Nil => complete(s"The color is '$color' and the city is $city.")
      case multiple    => complete(s"The color is '$color' and the cities are ${multiple.mkString
    }
  }

// tests:
Get("/?color=blue") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and there are no cities."
}
```

```
Get("/?color=blue&city=Chicago") ~> Route.seal(route) ~> check {
  responseAs[String] === "The color is 'blue' and the city is Chicago."
}

Get("/?color=blue&city=Chicago&city=Boston") ~> Route.seal(route) ~> check {
  responseAs[String] === "The color is 'blue' and the cities are Chicago, Boston."
}
val route =
  parameters('color, 'distance.as[Int].*) { (color, cities) =>
    cities.toList match {
      case Nil             => complete(s"The color is '$color' and there are no distances.")
      case distance :: Nil => complete(s"The color is '$color' and the distance is $distance.")
      case multiple        => complete(s"The color is '$color' and the distances are ${multiple.m
    }
  }

// tests:
Get("/?color=blue") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and there are no distances."
}

Get("/?color=blue&distance=5") ~> Route.seal(route) ~> check {
  responseAs[String] === "The color is 'blue' and the distance is 5."
}

Get("/?color=blue&distance=5&distance=14") ~> Route.seal(route) ~> check {
  responseAs[String] === "The color is 'blue' and the distances are 5, 14."
}
```

CSV parameter

```
val route =
  parameter("names".as(CsvSeq[String])) { names =>
    complete(s"The parameters are ${names.mkString(", ")}")
  }

// tests:
Get("/?names=") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are "
}
Get("/?names=Caplin") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are Caplin"
}
Get("/?names=Caplin,John") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are Caplin, John"
}
```

Repeated, deserialized parameter

```
val route =
  parameters('color, 'distance.as[Int].*) { (color, cities) =>
    cities.toList match {
      case Nil             => complete(s"The color is '$color' and there are no distances.")
      case distance :: Nil => complete(s"The color is '$color' and the distance is $distance.")
      case multiple        => complete(s"The color is '$color' and the distances are ${multiple.m
    }
  }

// tests:
Get("/?color=blue") ~> route ~> check {
  responseAs[String] === "The color is 'blue' and there are no distances."
}
```

```
Get("/?color=blue&distance=5") ~> Route.seal(route) ~> check {
  responseAs[String] === "The color is 'blue' and the distance is 5."
}

Get("/?color=blue&distance=5&distance=14") ~> Route.seal(route) ~> check {
  responseAs[String] === "The color is 'blue' and the distances are 5, 14."
}
```

parameterSeq

Signature

```
def parameterSeq: Directive1[immutable.Seq[(String, String)]]
```

Description Extracts all parameters at once in the original order as (name, value) tuples of type `(String, String)`.

This directive can be used if the exact order of parameters is important or if parameters can occur several times.

See *When to use which parameter directive?* to understand when to use which directive.

Example

```
val route =
  parameterSeq { params =>
    def paramString(param: (String, String)): String = s"""${param._1} = '${param._2}'"""
    complete(s"The parameters are ${params.map(paramString).mkString(", ")}")
  }

// tests:
Get("/?color=blue&count=42") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are color = 'blue', count = '42'"
}
Get("/?x=1&x=2") ~> route ~> check {
  responseAs[String] shouldEqual "The parameters are x = '1', x = '2'"
}
```

When to use which parameter directive? Usually, you want to use the high-level *parameters* directive. When you need more low-level access you can use the table below to decide which directive to use which shows properties of different parameter directives.

directive	level	ordering	multi
parameter	high	no	no
parameters	high	no	yes
parameterMap	low	no	no
parameterMultiMap	low	no	yes
parameterSeq	low	yes	yes

level high-level parameter directives extract subset of all parameters by name and allow conversions and automatically report errors if expectations are not met, low-level directives give you all parameters at once, leaving all further processing to you

ordering original ordering from request URL is preserved

multi multiple values per parameter name are possible

PathDirectives

path

Signature

```
def path[L](pm: PathMatcher[L]): Directive[L]
```

Description 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).

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")
        }
    }

// tests:
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

Signature

```
def pathEnd: Directive0
```

Description 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.

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")
      }
  }

// tests:
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

Signature

```
def pathEndOrSingleSlash: Directive0
```

Description Only passes the request to its inner route if the unmatched path of the `RequestContext` is either empty or contains only one single slash.

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")
      }
  }
```

```
// tests:
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

Signature

```
def pathPrefix[L](pm: PathMatcher[L]): Directive[L]
```

Description 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).

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")
      }
  }

// tests:
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

Signature

```
def pathPrefixTest[L](pm: PathMatcher[L]): Directive[L]
```

Description 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.

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 }
  }

// tests:
Get("/foo/doo") ~> route ~> check {
  responseAs[String] shouldEqual "/doo"
}

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

pathSingleSlash

Signature

```
def pathSingleSlash: Directive0
```

Description Only passes the request to its inner route if the unmatched path of the `RequestContext` contains exactly one single slash.

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")
        }
    }

// tests:
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

Signature

```
def pathSuffix[L](pm: PathMatcher[L]): Directive[L]
```

Description 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).

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
      }
  }

// tests:
Get("/start/middle/end") ~> route ~> check {
  responseAs[String] shouldEqual "/middle/"
}

Get("/start/something/barbaz/foo") ~> route ~> check {
  responseAs[String] shouldEqual "/something/"
}
```

pathSuffixTest

Signature

```
def pathSuffixTest[L](pm: PathMatcher[L]): Directive[L]
```

Description 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.

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")

// tests:
Get("/foo/") ~> route ~> check {
  responseAs[String] shouldEqual "slashed"
}
Get("/foo") ~> route ~> check {
  responseAs[String] shouldEqual "unslashed"
}
```

rawPathPrefix

Signature

```
def rawPathPrefix[L](pm: PathMatcher[L]): Directive[L]
```

Description 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).

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 }
  }

// tests:
Get("/foobar/baz") ~> route ~> check {
  responseAs[String] shouldEqual "/baz"
}

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

rawPathPrefixTest

Signature

```
def rawPathPrefixTest[L](pm: PathMatcher[L]): Directive[L]
```

Description 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.

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
    }
  }

// tests:
Get("/foobar") ~> route ~> check {
  responseAs[String] shouldEqual "bar"
}

Get("/foobaz") ~> route ~> check {
  handled shouldEqual false
}
```

redirectToNoTrailingSlashIfPresent

Signature

```
def redirectToNoTrailingSlashIfPresent(redirectionType: StatusCodes.Redirection): Directive0
```

Description If the requested path does end with a trailing `/` character, redirects to the same path without that trailing slash..

Redirects the HTTP Client to the same resource yet without the trailing `/`, in case the request contained it. When redirecting an HttpResponse with the given redirect response code (i.e. `MovedPermanently` or `TemporaryRedirect` etc.) as well as a simple HTML page containing a "*click me to follow redirect*" link to be used in case the client can not, or refuses to for security reasons, automatically follow redirects.

Please note that the inner paths **MUST NOT** end with an explicit trailing slash (e.g. `"things"./`) for the re-directed-to route to match.

A good read on the subject of how to deal with trailing slashes is available on Google Webmaster Central - To Slash or not to Slash.

See also *redirectToTrailingSlashIfMissing* for the opposite behaviour.

Example

```
import akka.http.scaladsl.model.StatusCodes

val route =
  redirectToNoTrailingSlashIfPresent(StatusCodes.MovedPermanently) {
    path("foo") {
      // We require the explicit trailing slash in the path
      complete("OK")
    } ~
      path("bad"./) {
        // MISTAKE!
        // Since inside a `redirectToNoTrailingSlashIfPresent` directive
        // the matched path here will never contain a trailing slash,
        // thus this path will never match.
        //
        // It should be `path("bad")` instead.
```

```
        ???
      }
  }

// tests:
// Redirected:
Get("/foo/") ~> route ~> check {
  status shouldEqual StatusCodes.MovedPermanently

  // results in nice human readable message,
  // in case the redirect can't be followed automatically:
  responseAs[String] shouldEqual {
    "This and all future requests should be directed to " +
      "<a href=\"http://example.com/foo\">this URI</a>."
  }
}

// Properly handled:
Get("/foo") ~> route ~> check {
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "OK"
}

// MISTAKE! will never match - reason explained in routes
Get("/bad") ~> route ~> check {
  handled shouldEqual false
}
```

See also *redirectToTrailingSlashIfMissing* which achieves the opposite - redirecting paths in case they do *not* have a trailing slash.

redirectToTrailingSlashIfMissing

Signature

```
def redirectToTrailingSlashIfMissing(redirectionType: StatusCodes.Redirection): Directive0
```

Description If the requested path does not end with a trailing `/` character, redirects to the same path followed by such trailing slash.

Redirects the HTTP Client to the same resource yet followed by a trailing `/`, in case the request did not contain it. When redirecting an HttpResponse with the given redirect response code (i.e. `MovedPermanently` or `TemporaryRedirect` etc.) as well as a simple HTML page containing a "*click me to follow redirect*" link to be used in case the client can not, or refuses to for security reasons, automatically follow redirects.

Please note that the inner paths **MUST** end with an explicit trailing slash (e.g. `"things"./`) for the re-directed-to route to match.

See also *redirectToNoTrailingSlashIfPresent* for the opposite behaviour.

Example

```
import akka.http.scaladsl.model.StatusCodes

val route =
  redirectToTrailingSlashIfMissing(StatusCodes.MovedPermanently) {
    path("foo"./) {
      // We require the explicit trailing slash in the path
      complete("OK")
    } ~
      path("bad-1") {
```

```
        // MISTAKE!
        // Missing `/` in path, causes this path to never match,
        // because it is inside a `redirectToTrailingSlashIfMissing`
        ???
      } ~
      path("bad-2/") {
        // MISTAKE!
        // / should be explicit as path element separator and not *in* the path element
        // So it should be: "bad-1" /
        ???
      }
  }

// tests:
// Redirected:
Get("/foo") ~> route ~> check {
  status shouldEqual StatusCodes.MovedPermanently

  // results in nice human readable message,
  // in case the redirect can't be followed automatically:
  responseAs[String] shouldEqual {
    "This and all future requests should be directed to " +
      "<a href=\"http://example.com/foo/\">this URI</a>."
  }
}

// Properly handled:
Get("/foo/") ~> route ~> check {
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "OK"
}

// MISTAKE! will never match - reason explained in routes
Get("/bad-1/") ~> route ~> check {
  handled shouldEqual false
}

// MISTAKE! will never match - reason explained in routes
Get("/bad-2/") ~> route ~> check {
  handled shouldEqual false
}
```

See also *redirectToNoTrailingSlashIfPresent* which achieves the opposite - redirecting paths in case they do have a trailing slash.

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* or *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 (unsigned) 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 (unsigned) 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")
```

RangeDirectives

withRangeSupport

Signature

```
def withRangeSupport(): Directive0
def withRangeSupport(rangeCountLimit: Int, rangeCoalescingThreshold:Long): Directive0
```

The signature shown is simplified, the real signature uses magnets. [10]

Description Transforms the response from its inner route into a `206 Partial Content` response if the client requested only part of the resource with a `Range` header.

Augments responses to `GET` requests with an `Accept-Ranges: bytes` header and converts them into partial responses if the request contains a valid `Range` request header. The requested byte-ranges are coalesced (merged) if they lie closer together than the specified `rangeCoalescingThreshold` argument.

In order to prevent the server from becoming overloaded with trying to prepare `multipart/byteranges` responses for high numbers of potentially very small ranges the directive rejects requests requesting more than `rangeCountLimit` ranges with a `TooManyRangesRejection`. Requests with unsatisfiable ranges are rejected with an `UnsatisfiableRangeRejection`.

The `withRangeSupport()` form (without parameters) uses the `range-coalescing-threshold` and `range-count-limit` settings from the `akka.http.routing` configuration.

This directive is transparent to non-`GET` requests.

See also: https://tools.ietf.org/html/rfc7233

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

Example

```
val route =
  withRangeSupport {
    complete("ABCDEFGH")
  }

Get() ~> addHeader(Range(ByteRange(3, 4))) ~> route ~> check {
  headers should contain(`Content-Range`(ContentRange(3, 4, 8)))
  status shouldEqual StatusCodes.PartialContent
  responseAs[String] shouldEqual "DE"
}

// we set "akka.http.routing.range-coalescing-threshold = 2"
// above to make sure we get two BodyParts
Get() ~> addHeader(Range(ByteRange(0, 1), ByteRange(1, 2), ByteRange(6, 7))) ~> route ~> check {
  headers.collectFirst { case `Content-Range`(_, _) => true } shouldBe None
  val responseF = responseAs[Multipart.ByteRanges].parts
    .runFold[List[Multipart.ByteRanges.BodyPart]](Nil)((acc, curr) => curr :: acc)

  val response = Await.result(responseF, 3.seconds).reverse

  response should have length 2

  val part1 = response(0)
  part1.contentRange === ContentRange(0, 2, 8)
  part1.entity should matchPattern {
    case HttpEntity.Strict(_, bytes) if bytes.utf8String == "ABC" =>
  }

  val part2 = response(1)
  part2.contentRange === ContentRange(6, 7, 8)
  part2.entity should matchPattern {
    case HttpEntity.Strict(_, bytes) if bytes.utf8String == "GH" =>
  }
}
```

RespondWithDirectives

respondWithDefaultHeader

Signature

```
def respondWithDefaultHeader(responseHeader: HttpHeader): Directive0
```

Description Adds a given HTTP header to all responses coming back from its inner route only if a header with the same name doesn't exist yet in the response.

This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by potentially adding the given `HttpHeader` instance to the headers list. The header is only added if there is no header instance with the same name (case insensitively) already present in the response.

See also *respondWithDefaultHeaders* if you'd like to add more than one header.

Example

```
// custom headers
val blippy = RawHeader("X-Fish-Name", "Blippy")
val elTonno = RawHeader("X-Fish-Name", "El Tonno")

// format: OFF
```

```
// by default always include the Blippy header,
// unless a more specific X-Fish-Name is given by the inner route
val route =
  respondWithDefaultHeader(blippy) {  //  blippy
    respondWithHeader(elTonno) {      // /  el tonno
      path("el-tonno") {              // | /
        complete("¡Ay blippy!")       // | |- el tonno
      } ~                             // | |
      path("los-tonnos") {            // | |
        complete("¡Ay ay blippy!")    // | |- el tonno
      }                               // | |
    } ~                               // | x
    complete("Blip!")                 // |- blippy
  } // x
// format: ON

// tests:
Get("/") ~> route ~> check {
  header("X-Fish-Name") shouldEqual Some(RawHeader("X-Fish-Name", "Blippy"))
  responseAs[String] shouldEqual "Blip!"
}

Get("/el-tonno") ~> route ~> check {
  header("X-Fish-Name") shouldEqual Some(RawHeader("X-Fish-Name", "El Tonno"))
  responseAs[String] shouldEqual "¡Ay blippy!"
}

Get("/los-tonnos") ~> route ~> check {
  header("X-Fish-Name") shouldEqual Some(RawHeader("X-Fish-Name", "El Tonno"))
  responseAs[String] shouldEqual "¡Ay ay blippy!"
}
```

respondWithDefaultHeaders

Signature

```
def respondWithDefaultHeaders(responseHeaders: HttpHeader*): Directive0
def respondWithDefaultHeaders(responseHeaders: immutable.Seq[HttpHeader]): Directive0
```

Description Adds the given HTTP headers to all responses coming back from its inner route only if a respective header with the same name doesn't exist yet in the response.

This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by potentially adding the given `HttpHeader` instances to the headers list. A header is only added if there is no header instance with the same name (case insensitively) already present in the response.

See also *respondWithDefaultHeader* if you'd like to add only a single header.

Example The `respondWithDefaultHeaders` directive is equivalent to the `respondWithDefaultHeader` directive which is shown in the example below, however it allows including multiple default headers at once in the directive, like so:

```
respondWithDefaultHeaders(
  Origin(HttpOrigin("http://akka.io"),
  RawHeader("X-Fish-Name", "Blippy"))) { /*...*/ }
```

The semantics remain the same however, as explained by the following example:

```
// custom headers
val blippy = RawHeader("X-Fish-Name", "Blippy")
```

```
val elTonno = RawHeader("X-Fish-Name", "El Tonno")

// format: OFF
// by default always include the Blippy header,
// unless a more specific X-Fish-Name is given by the inner route
val route =
  respondWithDefaultHeader(blippy) {  //  blippy
    respondWithHeader(elTonno) {      // /  el tonno
      path("el-tonno") {              // | /
        complete("¡Ay blippy!")       // | |- el tonno
      } ~                             // | |
      path("los-tonnos") {            // | |
        complete("¡Ay ay blippy!")    // | |- el tonno
      }                               // | |
    } ~                               // | x
    complete("Blip!")                 // |- blippy
  } // x
// format: ON

// tests:
Get("/") ~> route ~> check {
  header("X-Fish-Name") shouldEqual Some(RawHeader("X-Fish-Name", "Blippy"))
  responseAs[String] shouldEqual "Blip!"
}

Get("/el-tonno") ~> route ~> check {
  header("X-Fish-Name") shouldEqual Some(RawHeader("X-Fish-Name", "El Tonno"))
  responseAs[String] shouldEqual "¡Ay blippy!"
}

Get("/los-tonnos") ~> route ~> check {
  header("X-Fish-Name") shouldEqual Some(RawHeader("X-Fish-Name", "El Tonno"))
  responseAs[String] shouldEqual "¡Ay ay blippy!"
}
```

See the *respondWithDefaultHeader* directive for an example with only one header.

respondWithHeader

Signature

```
def respondWithHeader(responseHeader: HttpHeader): Directive0
```

Description Adds a given HTTP header to all responses coming back from its inner route.

This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by adding the given `HttpHeader` instance to the headers list.

See also *respondWithHeaders* if you'd like to add more than one header.

Example

```
val route =
  path("foo") {
    respondWithHeader(RawHeader("Funky-Muppet", "gonzo")) {
      complete("beep")
    }
  }

// tests:
Get("/foo") ~> route ~> check {
```

```
  header("Funky-Muppet") shouldEqual Some(RawHeader("Funky-Muppet", "gonzo"))
  responseAs[String] shouldEqual "beep"
}
```

respondWithHeaders

Signature

```
def respondWithHeaders(responseHeaders: HttpHeader*): Directive0
def respondWithHeaders(responseHeaders: immutable.Seq[HttpHeader]): Directive0
```

Description Adds the given HTTP headers to all responses coming back from its inner route.

This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by adding the given `HttpHeader` instances to the headers list.

See also *respondWithHeader* if you'd like to add just a single header.

Example

```
val route =
  path("foo") {
    respondWithHeaders(RawHeader("Funky-Muppet", "gonzo"), Origin(HttpOrigin("http://akka.io")))
      complete("beep")
    }
  }

// tests:
Get("/foo") ~> route ~> check {
  header("Funky-Muppet") shouldEqual Some(RawHeader("Funky-Muppet", "gonzo"))
  header[Origin] shouldEqual Some(Origin(HttpOrigin("http://akka.io")))
  responseAs[String] shouldEqual "beep"
}
```

respondWithHeaders

Signature

```
def respondWithHeaders(responseHeaders: HttpHeader*): Directive0
def respondWithHeaders(responseHeaders: immutable.Seq[HttpHeader]): Directive0
```

Description Adds the given HTTP headers to all responses coming back from its inner route.

This directive transforms `HttpResponse` and `ChunkedResponseStart` messages coming back from its inner route by adding the given `HttpHeader` instances to the headers list.

See also *respondWithHeader* if you'd like to add just a single header.

Example

```
val route =
  path("foo") {
    respondWithHeaders(RawHeader("Funky-Muppet", "gonzo"), Origin(HttpOrigin("http://akka.io")))
      complete("beep")
    }
  }

// tests:
Get("/foo") ~> route ~> check {
```

```
  header("Funky-Muppet") shouldEqual Some(RawHeader("Funky-Muppet", "gonzo"))
  header[Origin] shouldEqual Some(Origin(HttpOrigin("http://akka.io")))
  responseAs[String] shouldEqual "beep"
}
```

RouteDirectives The `RouteDirectives` have a special role in akka-http's routing DSL. Contrary to all other directives (except most *FileAndResourceDirectives*) they do not produce instances of type `Directive[L <: HList]` but rather "plain" routes of type `Route`. The reason is that the `RouteDirectives` 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 leaves of 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*):

complete

Signature

```
def complete[T :ToResponseMarshaller](value: T): StandardRoute
def complete(response: HttpResponse): StandardRoute
def complete(status: StatusCode): StandardRoute
def complete[T :Marshaller](status: StatusCode, value: T): StandardRoute
def complete[T :Marshaller](status: Int, value: T): StandardRoute
def complete[T :Marshaller](status: StatusCode, headers: Seq[HttpHeader], value: T): StandardRoute
def complete[T :Marshaller](status: Int, headers: Seq[HttpHeader], value: T): StandardRoute
```

The signature shown is simplified, the real signature uses magnets. [11]

Description Completes the request using the given argument(s).

`complete` uses the given arguments to construct a `Route` which simply calls `complete` on the `RequestContext` with the respective `HttpResponse` instance. Completing the request will send the response "back up" the route structure where all the logic runs that wrapping directives have potentially chained into the *RouteResult* future transformation chain.

Example

```
val route =
  path("a") {
    complete(HttpResponse(entity = "foo"))
  } ~
    path("b") {
      complete((StatusCodes.Created, "bar"))
    } ~
    (path("c") & complete("baz")) // `&` also works with `complete` as the 2nd argument

// tests:
Get("/a") ~> route ~> check {
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "foo"
}

Get("/b") ~> route ~> check {
  status shouldEqual StatusCodes.Created
  responseAs[String] shouldEqual "bar"
}

Get("/c") ~> route ~> check {
```

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

```
  status shouldEqual StatusCodes.OK
  responseAs[String] shouldEqual "baz"
}
```

failWith

Signature

```
def failWith(error: Throwable): StandardRoute
```

Description Bubbles up the given error through the route structure where it is dealt with by the closest `handleExceptions` directive and its `ExceptionHandler`.

`failWith` explicitly raises an exception that gets bubbled up through the route structure to be picked up by the nearest `handleExceptions` directive. Using `failWith` rather than simply throwing an exception enables the route structure's *Exception Handling* mechanism to deal with the exception even if the current route is executed asynchronously on another thread (e.g. in a `Future` or separate actor).

If no `handleExceptions` is present above the respective location in the route structure the top-level routing logic will handle the exception and translate it into a corresponding `HttpResponse` using the in-scope `ExceptionHandler` (see also the *Exception Handling* chapter).

There is one notable special case: If the given exception is a `RejectionError` exception it is *not* bubbled up, but rather the wrapped exception is unpacked and "executed". This allows the "tunneling" of a rejection via an exception.

Example

```
val route =
  path("foo") {
    failWith(new RuntimeException("Oops."))
  }

// tests:
Get("/foo") ~> Route.seal(route) ~> check {
  status shouldEqual StatusCodes.InternalServerError
  responseAs[String] shouldEqual "There was an internal server error."
}
```

redirect

Signature

```
def redirect(uri: Uri, redirectionType: Redirection): StandardRoute
```

Description Completes the request with a redirection response to a given targer URI and of a given redirection type (status code).

`redirect` is a convenience helper for completing the request with a redirection response. It is equivalent to this snippet relying on the `complete` directive:

```
omplete {
 HttpResponse(
   status = redirectionType,
   headers = headers.Location(uri) :: Nil,
   entity = redirectionType.htmlTemplate match {
     case ""       ⇒ HttpEntity.Empty
     case template ⇒ HttpEntity(ContentTypes.`text/html(UTF-8)`, template format uri)
   })
```

Example

```
val route =
  pathPrefix("foo") {
    pathSingleSlash {
      complete("yes")
    } ~
      pathEnd {
        redirect("/foo/", StatusCodes.PermanentRedirect)
      }
  }

// tests:
Get("/foo/") ~> route ~> check {
  responseAs[String] shouldEqual "yes"
}

Get("/foo") ~> route ~> check {
  status shouldEqual StatusCodes.PermanentRedirect
  responseAs[String] shouldEqual """The request, and all future requests should be repeated using
}
```

reject

Signature

```
def reject: StandardRoute
def reject(rejections: Rejection*): StandardRoute
```

Description Explicitly rejects the request optionally using the given rejection(s).

`reject` uses the given rejection instances (which might be the empty `Seq`) to construct a `Route` which simply calls `requestContext.reject`. See the chapter on *Rejections* for more information on what this means.

After the request has been rejected at the respective point it will continue to flow through the routing structure in the search for a route that is able to complete it.

The explicit `reject` directive is used mostly when building *Custom Directives*, e.g. inside of a `flatMap` modifier for "filtering out" certain cases.

Example

```
val route =
  path("a") {
    reject // don't handle here, continue on
  } ~
    path("a") {
      complete("foo")
    } ~
    path("b") {
      // trigger a ValidationRejection explicitly
      // rather than through the `validate` directive
      reject(ValidationRejection("Restricted!"))
    }

// tests:
Get("/a") ~> route ~> check {
  responseAs[String] shouldEqual "foo"
}

Get("/b") ~> route ~> check {
```

```
  rejection shouldEqual ValidationRejection("Restricted!")
}
```

SchemeDirectives Scheme directives can be used to extract the Uri scheme (i.e. "http", "https", etc.) from requests or to reject any request that does not match a specified scheme name.

extractScheme

Signature

```
def extractScheme: Directive1[String]
```

Description Extracts the Uri scheme (i.e. "`http`", "`https`", etc.) for an incoming request.

For rejecting a request if it doesn't match a specified scheme name, see the *scheme* directive.

Example

```
val route =
  extractScheme { scheme =>
    complete(s"The scheme is '${scheme}'")
  }

// tests:
Get("https://www.example.com/") ~> route ~> check {
  responseAs[String] shouldEqual "The scheme is 'https'"
}
```

scheme

Signature

```
def scheme(name: String): Directive0
```

Description Rejects a request if its Uri scheme does not match a given one.

The `scheme` directive can be used to match requests by their Uri scheme, only passing through requests that match the specified scheme and rejecting all others.

A typical use case for the `scheme` directive would be to reject requests coming in over http instead of https, or to redirect such requests to the matching https URI with a `MovedPermanently`.

For simply extracting the scheme name, see the *extractScheme* directive.

Example

```
import akka.http.scaladsl.model._
import akka.http.scaladsl.model.headers.Location
import StatusCodes.MovedPermanently

val route =
  scheme("http") {
    extract(_.request.uri) { uri ⇒
      redirect(uri.copy(scheme = "https"), MovedPermanently)
    }
  } ~
    scheme("https") {
      complete(s"Safe and secure!")
```

```
    }

// tests:
Get("http://www.example.com/hello") ~> route ~> check {
  status shouldEqual MovedPermanently
  header[Location] shouldEqual Some(Location(Uri("https://www.example.com/hello")))
}

Get("https://www.example.com/hello") ~> route ~> check {
  responseAs[String] shouldEqual "Safe and secure!"
}
```

SecurityDirectives

authenticateBasic

Signature

```
type Authenticator[T] = Credentials ⇒ Option[T]
```

```
def authenticateBasic[T](realm: String, authenticator: Authenticator[T]): AuthenticationDirective
```

Description Wraps the inner route with Http Basic authentication support using a given `Authenticator[T]`.

Provides support for handling HTTP Basic Authentication.

Given a function returning `Some[T]` upon successful authentication and `None` otherwise, respectively applies the inner route or rejects the request with a `AuthenticationFailedRejection` rejection, which by default is mapped to an `401 Unauthorized` response.

Longer-running authentication tasks (like looking up credentials in a database) should use the *authenticateBasicAsync* variant of this directive which allows it to run without blocking routing layer of Akka HTTP, freeing it for other requests.

Standard HTTP-based authentication which uses the `WWW-Authenticate` header containing challenge data and `Authorization` header for receiving credentials is implemented in subclasses of `HttpAuthenticator`.

See *Credentials and password timing attacks* for details about verifying the secret.

> **Warning:** Make sure to use basic authentication only over SSL/TLS because credentials are transferred in plaintext.

Example

```
def myUserPassAuthenticator(credentials: Credentials): Option[String] =
  credentials match {
    case p @ Credentials.Provided(id) if p.verify("p4ssw0rd") => Some(id)
    case _ => None
  }

val route =
  Route.seal {
    path("secured") {
      authenticateBasic(realm = "secure site", myUserPassAuthenticator) { userName =>
        complete(s"The user is '$userName'")
      }
    }
  }
```

```
// tests:
Get("/secured") ~> route ~> check {
  status shouldEqual StatusCodes.Unauthorized
  responseAs[String] shouldEqual "The resource requires authentication, which was not supplied wit
  header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("Basic", "secure site")
}

val validCredentials = BasicHttpCredentials("John", "p4ssw0rd")
Get("/secured") ~> addCredentials(validCredentials) ~> // adds Authorization header
  route ~> check {
    responseAs[String] shouldEqual "The user is 'John'"
  }

val invalidCredentials = BasicHttpCredentials("Peter", "pan")
Get("/secured") ~>
  addCredentials(invalidCredentials) ~> // adds Authorization header
  route ~> check {
    status shouldEqual StatusCodes.Unauthorized
    responseAs[String] shouldEqual "The supplied authentication is invalid"
    header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("Basic", "secure site
  }
```

authenticateBasicAsync

Signature

```
type AsyncAuthenticator[T] = Credentials ⇒ Future[Option[T]]
```

```
def authenticateBasicAsync[T](realm: String, authenticator: AsyncAuthenticator[T]): Authentication
```

Description Wraps the inner route with Http Basic authentication support using a given `AsyncAuthenticator[T]`.

This variant of the *authenticateBasic* directive returns a `Future[Option[T]]` which allows freeing up the routing layer of Akka HTTP, freeing it for other requests. It should be used whenever an authentication is expected to take a longer amount of time (e.g. looking up the user in a database).

In case the returned option is `None` the request is rejected with a `AuthenticationFailedRejection`, which by default is mapped to an `401 Unauthorized` response.

Standard HTTP-based authentication which uses the `WWW-Authenticate` header containing challenge data and `Authorization` header for receiving credentials is implemented in subclasses of `HttpAuthenticator`.

See *Credentials and password timing attacks* for details about verifying the secret.

> **Warning:** Make sure to use basic authentication only over SSL/TLS because credentials are transferred in plaintext.

Example

```
def myUserPassAuthenticator(credentials: Credentials): Future[Option[String]] =
  credentials match {
    case p @ Credentials.Provided(id) =>
      Future {
        // potentially
        if (p.verify("p4ssw0rd")) Some(id)
        else None
      }
    case _ => Future.successful(None)
  }
```

```
val route =
  Route.seal {
    path("secured") {
      authenticateBasicAsync(realm = "secure site", myUserPassAuthenticator) { userName =>
        complete(s"The user is '$userName'")
      }
    }
  }

// tests:
Get("/secured") ~> route ~> check {
  status shouldEqual StatusCodes.Unauthorized
  responseAs[String] shouldEqual "The resource requires authentication, which was not supplied wit
  header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("Basic", "secure site"
}

val validCredentials = BasicHttpCredentials("John", "p4ssw0rd")
Get("/secured") ~> addCredentials(validCredentials) ~> // adds Authorization header
  route ~> check {
    responseAs[String] shouldEqual "The user is 'John'"
  }

val invalidCredentials = BasicHttpCredentials("Peter", "pan")
Get("/secured") ~>
  addCredentials(invalidCredentials) ~> // adds Authorization header
  route ~> check {
    status shouldEqual StatusCodes.Unauthorized
    responseAs[String] shouldEqual "The supplied authentication is invalid"
    header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("Basic", "secure site
  }
```

authenticateBasicPF

Signature

```
type AuthenticatorPF[T] = PartialFunction[Credentials, T]

def authenticateBasicPF[T](realm: String, authenticator: AuthenticatorPF[T]): AuthenticationDirect
```

Description Wraps the inner route with Http Basic authentication support using a given `AuthenticatorPF[T]`.

Provides support for handling HTTP Basic Authentication.

Refer to *authenticateBasic* for a detailed description of this directive.

Its semantics are equivalent to `authenticateBasicPF` 's, where not handling a case in the Partial Function (PF) leaves the request to be rejected with a `AuthenticationFailedRejection` rejection.

Longer-running authentication tasks (like looking up credentials in a database) should use *authenticateBasicAsync* or *authenticateBasicPFAsync* if you prefer to use the `PartialFunction` syntax.

See *Credentials and password timing attacks* for details about verifying the secret.

> **Warning:** Make sure to use basic authentication only over SSL/TLS because credentials are transferred in plaintext.

Example

```
val myUserPassAuthenticator: AuthenticatorPF[String] = {
  case p @ Credentials.Provided(id) if p.verify("p4ssw0rd")         => id
  case p @ Credentials.Provided(id) if p.verify("p4ssw0rd-special") => s"$id-admin"
}

val route =
  Route.seal {
    path("secured") {
      authenticateBasicPF(realm = "secure site", myUserPassAuthenticator) { userName =>
        complete(s"The user is '$userName'")
      }
    }
  }

// tests:
Get("/secured") ~> route ~> check {
  status shouldEqual StatusCodes.Unauthorized
  responseAs[String] shouldEqual "The resource requires authentication, which was not supplied wit
  header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("Basic", "secure site"
}

val validCredentials = BasicHttpCredentials("John", "p4ssw0rd")
Get("/secured") ~> addCredentials(validCredentials) ~> // adds Authorization header
  route ~> check {
    responseAs[String] shouldEqual "The user is 'John'"
  }

val validAdminCredentials = BasicHttpCredentials("John", "p4ssw0rd-special")
Get("/secured") ~> addCredentials(validAdminCredentials) ~> // adds Authorization header
  route ~> check {
    responseAs[String] shouldEqual "The user is 'John-admin'"
  }

val invalidCredentials = BasicHttpCredentials("Peter", "pan")
Get("/secured") ~>
  addCredentials(invalidCredentials) ~> // adds Authorization header
  route ~> check {
    status shouldEqual StatusCodes.Unauthorized
    responseAs[String] shouldEqual "The supplied authentication is invalid"
    header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("Basic", "secure site
  }
```

authenticateBasicPFAsync

Signature

```
type AsyncAuthenticatorPF[T] = PartialFunction[Credentials, Future[T]]
```

```
def authenticateBasicPFAsync[T](realm: String, authenticator: AsyncAuthenticatorPF[T]): Authentica
```

Description Wraps the inner route with Http Basic authentication support using a given `AsyncAuthenticatorPF[T]`.

Provides support for handling HTTP Basic Authentication.

Refer to *authenticateBasic* for a detailed description of this directive.

Its semantics are equivalent to `authenticateBasicPF` 's, where not handling a case in the Partial Function (PF) leaves the request to be rejected with a `AuthenticationFailedRejection` rejection.

See *Credentials and password timing attacks* for details about verifying the secret.

Warning: Make sure to use basic authentication only over SSL/TLS because credentials are transferred in plaintext.

Example

```
case class User(id: String)
def fetchUser(id: String): Future[User] = {
  // some fancy logic to obtain a User
  Future.successful(User(id))
}

val myUserPassAuthenticator: AsyncAuthenticatorPF[User] = {
  case p @ Credentials.Provided(id) if p.verify("p4ssw0rd") =>
    fetchUser(id)
}

val route =
  Route.seal {
    path("secured") {
      authenticateBasicPFAsync(realm = "secure site", myUserPassAuthenticator) { user =>
        complete(s"The user is '${user.id}'")
      }
    }
  }

// tests:
Get("/secured") ~> route ~> check {
  status shouldEqual StatusCodes.Unauthorized
  responseAs[String] shouldEqual "The resource requires authentication, which was not supplied wit
  header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("Basic", "secure site")
}

val validCredentials = BasicHttpCredentials("John", "p4ssw0rd")
Get("/secured") ~> addCredentials(validCredentials) ~> // adds Authorization header
  route ~> check {
    responseAs[String] shouldEqual "The user is 'John'"
  }

val invalidCredentials = BasicHttpCredentials("Peter", "pan")
Get("/secured") ~>
  addCredentials(invalidCredentials) ~> // adds Authorization header
  route ~> check {
    status shouldEqual StatusCodes.Unauthorized
    responseAs[String] shouldEqual "The supplied authentication is invalid"
    header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("Basic", "secure site
  }
```

authenticateOrRejectWithChallenge

Signature

```
/**
 * The result of an HTTP authentication attempt is either the user object or
 * an HttpChallenge to present to the browser.
 */
type AuthenticationResult[+T] = Either[HttpChallenge, T]
```

```
def authenticateOrRejectWithChallenge[T](authenticator: Option[HttpCredentials] ⇒ Future[Authenti
def authenticateOrRejectWithChallenge[C <: HttpCredentials: ClassTag, T](
  authenticator: Option[C] ⇒ Future[AuthenticationResult[T]]): AuthenticationDirective[T]
```

Description Lifts an authenticator function into a directive.

This directive allows implementing the low level challange-response type of authentication that some services may require.

More details about challenge-response authentication are available in the RFC 2617, RFC 7616 and RFC 7617.

Example

```
val challenge = HttpChallenge("MyAuth", "MyRealm")

// your custom authentication logic:
def auth(creds: HttpCredentials): Boolean = true

def myUserPassAuthenticator(credentials: Option[HttpCredentials]): Future[AuthenticationResult[St
  Future {
    credentials match {
      case Some(creds) if auth(creds) => Right("some-user-name-from-creds")
      case _                          => Left(challenge)
    }
  }

val route =
  Route.seal {
    path("secured") {
      authenticateOrRejectWithChallenge(myUserPassAuthenticator _) { userName =>
        complete("Authenticated!")
      }
    }
  }

// tests:
Get("/secured") ~> route ~> check {
  status shouldEqual StatusCodes.Unauthorized
  responseAs[String] shouldEqual "The resource requires authentication, which was not supplied wit
  header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("MyAuth", "MyRealm")
}

val validCredentials = BasicHttpCredentials("John", "p4ssw0rd")
Get("/secured") ~> addCredentials(validCredentials) ~> // adds Authorization header
  route ~> check {
    status shouldEqual StatusCodes.OK
    responseAs[String] shouldEqual "Authenticated!"
  }
```

authenticateOAuth2

Signature

```
type Authenticator[T] = Credentials ⇒ Option[T]
```

```
def authenticateOAuth2[T](realm: String, authenticator: Authenticator[T]): AuthenticationDirective
```

Description Wraps the inner route with OAuth Bearer Token authentication support using a given `AuthenticatorPF[T]`

Provides support for extracting the so-called "*Bearer Token*" from the `Authorization` HTTP Header, which is used to initiate an OAuth2 authorization.

> **Warning:** This directive does not implement the complete OAuth2 protocol, but instead enables implementing it, by extracting the needed token from the HTTP headers.

Given a function returning `Some[T]` upon successful authentication and `None` otherwise, respectively applies the inner route or rejects the request with a `AuthenticationFailedRejection` rejection, which by default is mapped to an `401 Unauthorized` response.

Longer-running authentication tasks (like looking up credentials in a database) should use the *authenticateOAuth2Async* variant of this directive which allows it to run without blocking routing layer of Akka HTTP, freeing it for other requests.

See *Credentials and password timing attacks* for details about verifying the secret.

For more information on how OAuth2 works see RFC 6750.

Example Usage in code is exactly the same as *authenticateBasic*, with the difference that one must validate the token as OAuth2 dictates (which is currently not part of Akka HTTP itself).

authenticateOAuth2Async

Signature

```
type Authenticator[T] = Credentials ⇒ Option[T]
```

```
def authenticateOAuth2Async[T](realm: String, authenticator: AsyncAuthenticator[T]): Authenticati
```

Description Wraps the inner route with OAuth Bearer Token authentication support using a given `AsyncAuthenticator[T]`.

Provides support for extracting the so-called "*Bearer Token*" from the `Authorization` HTTP Header, which is used to initiate an OAuth2 authorization.

> **Warning:** This directive does not implement the complete OAuth2 protocol, but instead enables implementing it, by extracting the needed token from the HTTP headers.

Given a function returning `Some[T]` upon successful authentication and `None` otherwise, respectively applies the inner route or rejects the request with a `AuthenticationFailedRejection` rejection, which by default is mapped to an `401 Unauthorized` response.

See also *authenticateOAuth2* if the authorization operation is rather quick, and does not have to execute asynchronously.

See *Credentials and password timing attacks* for details about verifying the secret.

For more information on how OAuth2 works see RFC 6750.

Example Usage in code is exactly the same as *authenticateBasicAsync*, with the difference that one must validate the token as OAuth2 dictates (which is currently not part of Akka HTTP itself).

authenticateOAuth2PF

Signature

```
type Authenticator[T] = Credentials ⇒ Option[T]
```

```
def authenticateOAuth2PF[T](realm: String, authenticator: AuthenticatorPF[T]): AuthenticationDire
```

Description Wraps the inner route with OAuth Bearer Token authentication support using a given `AuthenticatorPF[T]`.

Provides support for extracting the so-called "*Bearer Token*" from the `Authorization` HTTP Header, which is used to initiate an OAuth2 authorization.

> **Warning:** This directive does not implement the complete OAuth2 protocol, but instead enables implementing it, by extracting the needed token from the HTTP headers.

Refer to *authenticateOAuth2* for a detailed description of this directive.

Its semantics are equivalent to `authenticateOAuth2PF` 's, where not handling a case in the Partial Function (PF) leaves the request to be rejected with a `AuthenticationFailedRejection` rejection.

Longer-running authentication tasks (like looking up credentials in a database) should use the *authenticateOAuth2Async* variant of this directive which allows it to run without blocking routing layer of Akka HTTP, freeing it for other requests.

See *Credentials and password timing attacks* for details about verifying the secret.

For more information on how OAuth2 works see RFC 6750.

Example Usage in code is exactly the same as *authenticateBasicPF*, with the difference that one must validate the token as OAuth2 dictates (which is currently not part of Akka HTTP itself).

authenticateOAuth2PFAsync Wraps the inner route with OAuth Bearer Token authentication support using a given `AsyncAuthenticatorPF[T]`.

Signature

```
type Authenticator[T] = Credentials ⇒ Option[T]
```

```
def authenticateOAuth2PFAsync[T](realm: String, authenticator: AsyncAuthenticatorPF[T]): Authentic
```

Description Provides support for extracting the so-called "*Bearer Token*" from the `Authorization` HTTP Header, which is used to initiate an OAuth2 authorization.

> **Warning:** This directive does not implement the complete OAuth2 protocol, but instead enables implementing it, by extracting the needed token from the HTTP headers.

Refer to *authenticateOAuth2* for a detailed description of this directive.

Its semantics are equivalent to `authenticateOAuth2PF` 's, where not handling a case in the Partial Function (PF) leaves the request to be rejected with a `AuthenticationFailedRejection` rejection.

See also *authenticateOAuth2PF* if the authorization operation is rather quick, and does not have to execute asynchronously.

See *Credentials and password timing attacks* for details about verifying the secret.

For more information on how OAuth2 works see RFC 6750.

Example Usage in code is exactly the same as *authenticateBasicPFAsync*, with the difference that one must validate the token as OAuth2 dictates (which is currently not part of Akka HTTP itself).

authenticateOrRejectWithChallenge

Signature

```
/**
 * The result of an HTTP authentication attempt is either the user object or
 * an HttpChallenge to present to the browser.
 */
type AuthenticationResult[+T] = Either[HttpChallenge, T]
```

```
def authenticateOrRejectWithChallenge[T](authenticator: Option[HttpCredentials] ⇒ Future[Authenti
def authenticateOrRejectWithChallenge[C <: HttpCredentials: ClassTag, T](
  authenticator: Option[C] ⇒ Future[AuthenticationResult[T]]): AuthenticationDirective[T]
```

Description Lifts an authenticator function into a directive.

This directive allows implementing the low level challange-response type of authentication that some services may require.

More details about challenge-response authentication are available in the RFC 2617, RFC 7616 and RFC 7617.

Example

```
val challenge = HttpChallenge("MyAuth", "MyRealm")

// your custom authentication logic:
def auth(creds: HttpCredentials): Boolean = true

def myUserPassAuthenticator(credentials: Option[HttpCredentials]): Future[AuthenticationResult[St
  Future {
    credentials match {
      case Some(creds) if auth(creds) => Right("some-user-name-from-creds")
      case _                          => Left(challenge)
    }
  }

val route =
  Route.seal {
    path("secured") {
      authenticateOrRejectWithChallenge(myUserPassAuthenticator _) { userName =>
        complete("Authenticated!")
      }
    }
  }

// tests:
Get("/secured") ~> route ~> check {
  status shouldEqual StatusCodes.Unauthorized
  responseAs[String] shouldEqual "The resource requires authentication, which was not supplied wit
  header[`WWW-Authenticate`].get.challenges.head shouldEqual HttpChallenge("MyAuth", "MyRealm")
}

val validCredentials = BasicHttpCredentials("John", "p4ssw0rd")
Get("/secured") ~> addCredentials(validCredentials) ~> // adds Authorization header
  route ~> check {
    status shouldEqual StatusCodes.OK
    responseAs[String] shouldEqual "Authenticated!"
  }
```

authorize

Signature

```
def authorize(check: ⇒ Boolean): Directive0
def authorize(check: RequestContext ⇒ Boolean): Directive0
```

Description Applies the given authorization check to the request.

The user-defined authorization check can either be supplied as a `=> Boolean` value which is calculated just from information out of the lexical scope, or as a function `RequestContext => Boolean` which can also take information from the request itself into account.

If the check returns `true` the request is passed on to the inner route unchanged, otherwise an `AuthorizationFailedRejection` is created, triggering a `403 Forbidden` response by default (the same as in the case of an `AuthenticationFailedRejection`).

In a common use-case you would check if a user (e.g. supplied by any of the `authenticate*` family of directives, e.g. *authenticateBasic*) is allowed to access the inner routes, e.g. by checking if the user has the needed permissions.

Note: See also *Authentication vs. Authorization* to understand the differences between those.

Example

```
case class User(name: String)

// authenticate the user:
def myUserPassAuthenticator(credentials: Credentials): Option[User] =
  credentials match {
    case Credentials.Provided(id) => Some(User(id))
    case _                        => None
  }

// check if user is authorized to perform admin actions:
val admins = Set("Peter")
def hasAdminPermissions(user: User): Boolean =
  admins.contains(user.name)

val route =
  Route.seal {
    authenticateBasic(realm = "secure site", myUserPassAuthenticator) { user =>
      path("peters-lair") {
        authorize(hasAdminPermissions(user)) {
          complete(s"'${user.name}' visited Peter's lair")
        }
      }
    }
  }

// tests:
val johnsCred = BasicHttpCredentials("John", "p4ssw0rd")
Get("/peters-lair") ~> addCredentials(johnsCred) ~> // adds Authorization header
  route ~> check {
    status shouldEqual StatusCodes.Forbidden
    responseAs[String] shouldEqual "The supplied authentication is not authorized to access this
  }

val petersCred = BasicHttpCredentials("Peter", "pan")
Get("/peters-lair") ~> addCredentials(petersCred) ~> // adds Authorization header
  route ~> check {
    responseAs[String] shouldEqual "'Peter' visited Peter's lair"
  }
```

extractCredentials

Signature

```
def extractCredentials: Directive1[Option[HttpCredentials]]
```

Description Extracts the potentially present `HttpCredentials` provided with the request's `Authorization` header, which can be then used to implement some custom authentication or authorization logic.

See *Credentials and password timing attacks* for details about verifying the secret.

Example

```
val route =
  extractCredentials { creds =>
    complete {
      creds match {
        case Some(c) => "Credentials: " + c
        case _       => "No credentials"
      }
    }
  }

// tests:
val johnsCred = BasicHttpCredentials("John", "p4ssw0rd")
Get("/") ~> addCredentials(johnsCred) ~> // adds Authorization header
  route ~> check {
    responseAs[String] shouldEqual "Credentials: Basic Sm9objpwNHNzdzByZA=="
  }

Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "No credentials"
}
```

Authentication vs. Authorization **Authentication** is the process of establishing a known identity for the user, whereby 'identity' is defined in the context of the application. This may be done with a username/password combination, a cookie, a pre-defined IP or some other mechanism. After authentication the system believes that it knows who the user is.

Authorization is the process of determining, whether a given user is allowed access to a given resource or not. In most cases, in order to be able to authorize a user (i.e. allow access to some part of the system) the users identity must already have been established, i.e. he/she must have been authenticated. Without prior authentication the authorization would have to be very crude, e.g. "allow access for *all* users" or "allow access for *noone*". Only after authentication will it be possible to, e.g., "allow access to the statistics resource for *admins*, but not for regular *members*".

Authentication and authorization may happen at the same time, e.g. when everyone who can properly be authenticated is also allowed access (which is often a very simple and somewhat implicit authorization logic). In other cases the system might have one mechanism for authentication (e.g. establishing user identity via an LDAP lookup) and another one for authorization (e.g. a database lookup for retrieving user access rights).

Authentication and Authorization in HTTP HTTP provides a general framework for access control and authentication, via an extensible set of challenge-response authentication schemes, which can be used by a server to challenge a client request and by a client to provide authentication information. The general mechanism is defined in RFC 7235.

The "HTTP Authentication Scheme Registry" defines the namespace for the authentication schemes in challenges and credentials. You can see the currently registered schemes at http://www.iana.org/assignments/http-authschemes.

At this point Akka HTTP only implements the "'Basic' HTTP Authentication Scheme" whose most current specification can be found here: https://datatracker.ietf.org/doc/draft-ietf-httpauth-basicauth-update/.

Low-level OAuth2 "Bearer Token" directives The OAuth2 directives currently provided in Akka HTTP are not a full OAuth2 protocol implementation, they are only a means of extracting the so called `Bearer Token` from the `Authorization` HTTP Header, as defined in RFC 6750, and allow users to validate and complete the protocol.

Credentials and password timing attacks When transforming request `Credentials` into an application specific user identifier the naive solution for checking the secret (password) would be a regular string comparison, but doing this would open up the application to timing attacks. See for example Timing Attacks Explained for an explanation of the problem.

To protect users of the library from that mistake the secret is not available through the API, instead the method `Credentials.Provided.verify(String)` should be used. It does a constant time comparison rather than returning early upon finding the first non-equal character.

WebsocketDirectives

handleWebsocketMessages

Signature

```
def handleWebsocketMessages(handler: Flow[Message, Message, Any]): Route
```

Description The directive first checks if the request was a valid Websocket handshake request and if yes, it completes the request with the passed handler. Otherwise, the request is rejected with an `ExpectedWebsocketRequestRejection`.

Websocket subprotocols offered in the `Sec-Websocket-Protocol` header of the request are ignored. If you want to support several protocols use the *handleWebsocketMessagesForProtocol* directive, instead.

For more information about the Websocket support, see *Server-Side WebSocket Support*.

Example

```
def greeter: Flow[Message, Message, Any] =
  Flow[Message].mapConcat {
    case tm: TextMessage ⇒
      TextMessage(Source.single("Hello ") ++ tm.textStream ++ Source.single("!")) :: Nil
    case bm: BinaryMessage ⇒
      // ignore binary messages but drain content to avoid the stream being clogged
      bm.dataStream.runWith(Sink.ignore)
      Nil
  }
val websocketRoute =
  path("greeter") {
    handleWebsocketMessages(greeter)
  }

// tests:
// create a testing probe representing the client-side
val wsClient = WSProbe()

// WS creates a Websocket request for testing
WS("/greeter", wsClient.flow) ~> websocketRoute ~>
  check {
    // check response for WS Upgrade headers
```

```
    isWebsocketUpgrade shouldEqual true

    // manually run a WS conversation
    wsClient.sendMessage("Peter")
    wsClient.expectMessage("Hello Peter!")

    wsClient.sendMessage(BinaryMessage(ByteString("abcdef")))
    wsClient.expectNoMessage(100.millis)

    wsClient.sendMessage("John")
    wsClient.expectMessage("Hello John!")

    wsClient.sendCompletion()
    wsClient.expectCompletion()
  }
```

handleWebsocketMessagesForProtocol

Signature

```
def handleWebsocketMessagesForProtocol(handler: Flow[Message, Message, Any], subprotocol: String)
```

Description Handles Websocket requests with the given handler if the given subprotocol is offered in the `Sec-Websocket-Protocol` header of the request and rejects other requests with an `ExpectedWebsocketRequestRejection` or an `UnsupportedWebsocketSubprotocolRejection`.

The directive first checks if the request was a valid Websocket handshake request and if the request offers the passed subprotocol name. If yes, the directive completes the request with the passed handler. Otherwise, the request is either rejected with an `ExpectedWebsocketRequestRejection` or an `UnsupportedWebsocketSubprotocolRejection`.

To support several subprotocols, for example at the same path, several instances of `handleWebsocketMessagesForProtocol` can be chained using `~` as you can see in the below example.

For more information about the Websocket support, see *Server-Side WebSocket Support*.

Example

```
def greeterService: Flow[Message, Message, Any] =
  Flow[Message].mapConcat {
    case tm: TextMessage ⇒
      TextMessage(Source.single("Hello ") ++ tm.textStream ++ Source.single("!")) :: Nil
    case bm: BinaryMessage ⇒
      // ignore binary messages but drain content to avoid the stream being clogged
      bm.dataStream.runWith(Sink.ignore)
      Nil
  }

def echoService: Flow[Message, Message, Any] =
  Flow[Message]
    // needed because a noop flow hasn't any buffer that would start processing in tests
    .buffer(1, OverflowStrategy.backpressure)

def websocketMultipleProtocolRoute =
  path("services") {
    handleWebsocketMessagesForProtocol(greeterService, "greeter") ~
      handleWebsocketMessagesForProtocol(echoService, "echo")
  }
```

```
// tests:
val wsClient = WSProbe()

// WS creates a Websocket request for testing
WS("/services", wsClient.flow, List("other", "echo")) ~>
  websocketMultipleProtocolRoute ~>
  check {
    expectWebsocketUpgradeWithProtocol { protocol ⇒
      protocol shouldEqual "echo"

      wsClient.sendMessage("Peter")
      wsClient.expectMessage("Peter")

      wsClient.sendMessage(BinaryMessage(ByteString("abcdef")))
      wsClient.expectMessage(ByteString("abcdef"))

      wsClient.sendMessage("John")
      wsClient.expectMessage("John")

      wsClient.sendCompletion()
      wsClient.expectCompletion()
    }
  }
```

Custom Directives

Part of the power of akka-http directives comes from the ease with which it's possible to define custom directives at differing levels of abstraction.

There are essentially three ways of creating custom directives:

1. By introducing new "labels" for configurations of existing directives
2. By transforming existing directives
3. By writing a directive "from scratch"

Configuration Labeling

The easiest way to create a custom directive is to simply assign a new name for a certain configuration of one or more existing directives. In fact, most of the predefined akka-http directives can be considered named configurations of more low-level directives.

The basic technique is explained in the chapter about Composing Directives, where, for example, a new directive `getOrPut` is defined like this:

```
val getOrPut = get | put

// tests:
val route = getOrPut { complete("ok") }

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

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

Another example is the *MethodDirectives* which are simply instances of a preconfigured *method* directive. The low-level directives that most often form the basis of higher-level "named configuration" directives are grouped

together in the *BasicDirectives* trait.

Transforming Directives

The second option for creating new directives is to transform an existing one using one of the "transformation methods", which are defined on the Directive class, the base class of all "regular" directives.

Apart from the combinator operators (`|` and `&`) and the case-class extractor (`as[T]`) there following transformations is also defined on all `Directive` instances:

- *map and tmap*
- *flatMap and tflatMap*
- *require and trequire*
- *recover and recoverPF*

map and tmap If the Directive is a single-value `Directive`, the `map` method allows for simple transformations:

```
val textParam: Directive1[String] =
  parameter("text".as[String])

val lengthDirective: Directive1[Int] =
  textParam.map(text => text.length)

// tests:
Get("/?text=abcdefg") ~> lengthDirective(x => complete(x.toString)) ~> check {
  responseAs[String] === "7"
}
```

One example of a predefined directive relying on `map` is the optionalHeaderValue directive.

The tmap modifier has this signature (somewhat simplified):

```
def tmap[R](f: L ⇒ R): Directive[Out]
```

It can be used to transform the `Tuple` of extractions into another `Tuple`. The number and/or types of the extractions can be changed arbitrarily. For example if `R` is `Tuple2[A, B]` then the result will be a `Directive[(A, B)]`. Here is a somewhat contrived example:

```
val twoIntParameters: Directive[(Int, Int)] =
  parameters(("a".as[Int], "b".as[Int]))

val myDirective: Directive1[String] =
  twoIntParameters.tmap {
    case (a, b) => (a + b).toString
  }

// tests:
Get("/?a=2&b=5") ~> myDirective(x => complete(x)) ~> check {
  responseAs[String] === "7"
}
```

flatMap and tflatMap With map and tmap you can transform the values a directive extracts but you cannot change the "extracting" nature of the directive. For example, if you have a directive extracting an `Int` you can use map to turn it into a directive that extracts that `Int` and doubles it, but you cannot transform it into a directive, that doubles all positive `Int` values and rejects all others.

In order to do the latter you need `flatMap` or `tflatMap`. The `tflatMap` modifier has this signature:

```
def tflatMap[R: Tuple](f: L ⇒ Directive[R]): Directive[R]
```

The given function produces a new directive depending on the Tuple of extractions of the underlying one. As in the case of *map and tmap* there is also a single-value variant called `flatMap`, which simplifies the operation for Directives only extracting one single value.

Here is the (contrived) example from above, which doubles positive Int values and rejects all others:

```
val intParameter: Directive1[Int] = parameter("a".as[Int])

val myDirective: Directive1[Int] =
  intParameter.flatMap {
    case a if a > 0 => provide(2 * a)
    case _          => reject
  }

// tests:
Get("/?a=21") ~> myDirective(i => complete(i.toString)) ~> check {
  responseAs[String] === "42"
}
Get("/?a=-18") ~> myDirective(i => complete(i.toString)) ~> check {
  handled === false
}
```

A common pattern that relies on flatMap is to first extract a value from the RequestContext with the extract directive and then flatMap with some kind of filtering logic. For example, this is the implementation of the method directive:

```
/**
 * Rejects all requests whose HTTP method does not match the given one.
 */
def method(httpMethod: HttpMethod): Directive0 =
  extractMethod.flatMap[Unit] {
    case `httpMethod` ⇒ pass
    case _            ⇒ reject(MethodRejection(httpMethod))
  } & cancelRejections(classOf[MethodRejection])
```

The explicit type parameter `[Unit]` on the flatMap i's needed in this case because the result of the flatMap is directly concatenated with the `cancelAllRejections` directive, thereby preventing "outside-in" inference of the type parameter value.

require and trequire The require modifier transforms a single-extraction directive into a directive without extractions, which filters the requests according the a predicate function. All requests, for which the predicate is false are rejected, all others pass unchanged.

The signature of require is this:

```
def require(predicate: T ⇒ Boolean, rejections: Rejection*): Directive0
```

One example of a predefined directive relying on require is the first overload of the host directive:

```
/**
 * Rejects all requests for whose host name the given predicate function returns false.
 */
def host(predicate: String ⇒ Boolean): Directive0 = extractHost.require(predicate)
```

You can only call require on single-extraction directives. The trequire modifier is the more general variant, which takes a predicate of type `Tuple => Boolean`. It can therefore also be used on directives with several extractions.

recover and recoverPF The `recover` modifier allows you "catch" rejections produced by the underlying directive and, instead of rejecting, produce an alternative directive with the same type(s) of extractions.

The signature of recover is this:

```
def recover[R >: L: Tuple](recovery: Seq[Rejection] ⇒ Directive[R]): Directive[R] =
```

In many cases the very similar `recoverPF` modifier might be little bit easier to use since it doesn't require the handling of all rejections:

```
def recoverPF[R >: L: Tuple](
  recovery: PartialFunction[Seq[Rejection], Directive[R]]): Directive[R]
```

One example of a predefined directive relying `recoverPF` is the optionalHeaderValue directive:

```
/**
 * Extracts an optional HTTP header value using the given function.
 * If the given function throws an exception the request is rejected
 * with a [[spray.routing.MalformedHeaderRejection]].
 */
def optionalHeaderValue[T](f: HttpHeader ⇒ Option[T]): Directive1[Option[T]] =
  headerValue(f).map(Some(_): Option[T]).recoverPF {
    case Nil ⇒ provide(None)
  }
```

Directives from Scratch

The third option for creating custom directives is to do it "from scratch", by directly subclassing the Directive class. The Directive is defined like this (leaving away operators and modifiers):

```
abstract class Directive[L](implicit val ev: Tuple[L]) {

  /**
   * Calls the inner route with a tuple of extracted values of type `L`.
   *
   * `tapply` is short for "tuple-apply". Usually, you will use the regular `apply` method instead
   * which is added by an implicit conversion (see `Directive.addDirectiveApply`).
   */
  def tapply(f: L ⇒ Route): Route
```

It only has one abstract member that you need to implement, the happly method, which creates the Route the directives presents to the outside from its inner Route building function (taking the extractions as parameter).

Extractions are kept as a Tuple. Here are a few examples:

A `Directive[Unit]` extracts nothing (like the get directive). Because this type is used quite frequently akka-http defines a type alias for it:

```
type Directive0 = Directive[Unit]
```

A `Directive[(String)]` extracts one String value (like the hostName directive). The type alias for it is:

```
type Directive1[T] = Directive[Tuple1[T]]
```

A Directive[(Int, String)] extracts an `Int` value and a `String` value (like a `parameters('a.as[Int], 'b.as[String])` directive).

Keeping extractions as `Tuples` has a lot of advantages, mainly great flexibility while upholding full type safety and "inferability". However, the number of times where you'll really have to fall back to defining a directive from scratch should be very small. In fact, if you find yourself in a position where a "from scratch" directive is your only option, we'd like to hear about it, so we can provide a higher-level "something" for other users.

Basics

Directives create *Routes*. To understand how directives work it is helpful to contrast them with the "primitive" way of creating routes.

Since `Route` is just a type alias for a function type `Route` instances can be written in any way in which function instances can be written, e.g. as a function literal:

```
val route: Route = { ctx => ctx.complete("yeah") }
```

or shorter:

```
val route: Route = _.complete("yeah")
```

With the *complete* directive this becomes even shorter:

```
val route = complete("yeah")
```

These three ways of writing this `Route` are fully equivalent, the created `route` will behave identically in all cases.

Let's look at a slightly more complicated example to highlight one important point in particular. Consider these two routes:

```
val a: Route = {
  println("MARK")
  ctx => ctx.complete("yeah")
}

val b: Route = { ctx =>
  println("MARK")
  ctx.complete("yeah")
}
```

The difference between `a` and `b` is when the `println` statement is executed. In the case of `a` it is executed *once*, when the route is constructed, whereas in the case of `b` it is executed every time the route is *run*.

Using the *complete* directive the same effects are achieved like this:

```
val a = {
  println("MARK")
  complete("yeah")
}

val b = complete {
  println("MARK")
  "yeah"
}
```

This works because the argument to the *complete* directive is evaluated *by-name*, i.e. it is re-evaluated every time the produced route is run.

Let's take things one step further:

```
val route: Route = { ctx =>
  if (ctx.request.method == HttpMethods.GET)
    ctx.complete("Received GET")
  else
    ctx.complete("Received something else")
}
```

Using the *get* and *complete* directives we can write this route like this:

```
val route =
  get {
    complete("Received GET")
  } ~
  complete("Received something else")
```

Again, the produced routes will behave identically in all cases.

Note that, if you wish, you can also mix the two styles of route creation:

```
val route =
  get { ctx =>
    ctx.complete("Received GET")
  } ~
  complete("Received something else")
```

Here, the inner route of the *get* directive is written as an explicit function literal.

However, as you can see from these examples, building routes with directives rather than "manually" results in code that is a lot more concise and as such more readable and maintainable. In addition it provides for better composability (as you will see in the coming sections). So, when using Akka HTTP's Routing DSL you should almost never have to fall back to creating routes via `Route` function literals that directly manipulate the *RequestContext*.

Structure

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 (The *RouteDirectives* are special in that they are always used at the leaf-level and as such cannot have inner routes). 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 (i.e. modify the request)
- Filter the `RequestContext` according to some logic, i.e. only pass on certain requests and reject others
- Extract values from the `RequestContext` and make them available to its inner route as "extractions"
- Chain some logic into the *RouteResult* future transformation chain (i.e. modify the response or rejection)
- Complete the request

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

Composing Directives

Note: Gotcha: forgetting the ~ (tilde) character in between directives can often result in perfectly valid Scala code that compiles but lead to your composed directive only containing the up to where ~ is missing.

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

```
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(s"Received ${ctx.request.method.name} request for order $id")
    }
  }
```

Or better (without dropping down to writing an explicit `Route` function manually):

```
val route =
  path("order" / IntNumber) { id =>
    (get | put) {
      extractMethod { m =>
        complete(s"Received ${m.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 {
      extractMethod { m =>
        complete(s"Received ${m.name} request for order $id")
      }
    }
  }
```

Note that, because `getOrPut` doesn't take any parameters, it can be a `val` here.

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

```
val getOrPut = get | put
val route =
  (path("order" / IntNumber) & getOrPut & extractMethod) { (id, m) =>
```

```
    complete(s"Received ${m.name} request for order $id")
  }
```

Here you can see that, when directives producing extractions are combined with `&`, the resulting "super-directive" simply extracts the concatenation of its sub-extractions.

And once again, you can factor things out if you want, thereby pushing the "factoring out" of directive configurations to its extreme:

```
val orderGetOrPutWithMethod =
  path("order" / IntNumber) & (get | put) & extractMethod
val route =
  orderGetOrPutWithMethod { (id, m) =>
    complete(s"Received ${m.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.

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*.

2.5.4 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 the many *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.

The 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 (immutable.Seq[Rejection] ⇒ Option[Route])
```

Since a `RejectionHandler` returns an `Option[Route]` it can choose whether it would like to handle the current set of rejections or not. If it returns `None` the rejections will simply continue to flow through the route structure.

The default `RejectionHandler` applied by the top-level glue code that turns a `Route` into a `Flow` or async handler function for the *low-level API* (via `Route.handlerFlow` or `Route.asyncHandler`) will handle *all* rejections that reach it.

Rejection Cancellation

As you can see from its definition above the `RejectionHandler` doesn't handle 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 compressed POST")
      }
    }
  }
```

For uncompressed POST requests this route structure would initially 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 *decodeRequestWith* directive (which only accepts gzip-compressed requests here)

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 a list (or rather immutable `Seq`) 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.

Customizing Rejection Handling

If you'd like to customize the way certain rejections are handled you'll have to write a custom *RejectionHandler*. Here is an example:

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.http.scaladsl.server._
import StatusCodes._
import Directives._

implicit def myRejectionHandler =
  RejectionHandler.newBuilder()
    .handle { case MissingCookieRejection(cookieName) =>
      complete(HttpResponse(BadRequest, entity = "No cookies, no service!!!"))
    }
    .handle { case AuthorizationFailedRejection ⇒
      complete((Forbidden, "You're out of your depth!"))
    }
    .handle { case ValidationRejection(msg, _) ⇒
      complete((InternalServerError, "That wasn't valid! " + msg))
    }
    .handleAll[MethodRejection] { methodRejections ⇒
      val names = methodRejections.map(_.supported.name)
      complete((MethodNotAllowed, s"Can't do that! Supported: ${names mkString " or "}!"))
    }
    .handleNotFound { complete((NotFound, "Not here!")) }
    .result()

object MyApp extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()

  val route: Route =
    // ... some route structure

  Http().bindAndHandle(route, "localhost", 8080)
}
```

The easiest way to construct a `RejectionHandler` is via the `RejectionHandler.Builder` that Akka HTTP provides. After having created a new `Builder` instance with `RejectionHandler.newBuilder()` you can attach handling logic for certain types of rejections through three helper methods:

handle Handles certain rejections with the given partial function. The partial function simply produces a `Route` which is run when the rejection is "caught". This makes the full power of the Routing DSL available for defining rejection handlers and even allows for recursing back into the main route structure if required.

handleAll[T <: Rejection] Handles all rejections of a certain type at the same time. This is useful for cases where your need access to more than the first rejection of a certain type, e.g. for producing the error message to an unsupported request method.

handleNotFound As described *above* "Resource Not Found" is special as it is represented with an empty rejection set. The `handleNotFound` helper let's you specify the "recovery route" for this case.

Even though you could handle several different rejection types in a single partial function supplied to `handle` it is recommended to split these up into distinct `handle` attachments instead. This way the priority between rejections is properly defined via the order of your `handle` clauses rather than the (sometimes hard to predict or control) order of rejections in the rejection set.

Once you have defined your custom `RejectionHandler` you have two options for "activating" it:

1. Bring it into implicit scope at the top-level.
2. Supply it as argument to the *handleRejections* directive.

In the first case your handler will be "sealed" (which means that it will receive the default handler as a fallback for all cases your handler doesn't handle itself) and used for all rejections that are not handled within the route structure itself.

The second case allows you to restrict the applicability of your handler to certain branches of your route structure.

2.5.5 Exception Handling

Exceptions thrown during route execution bubble up through the route structure to the next enclosing *handleExceptions* directive or the top of your route structure.

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

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

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. At the root of the route tree any still unhandled exception will be dealt with by the top-level handler which always handles *all* exceptions.

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

So, if you'd like to customize the way certain exceptions are handled you need to write a custom `ExceptionHandler`. Once you have defined your custom `ExceptionHandler` you have two options for "activating" it:

1. Bring it into implicit scope at the top-level.
2. Supply it as argument to the *handleExceptions* directive.

In the first case your handler will be "sealed" (which means that it will receive the default handler as a fallback for all cases your handler doesn't handle itself) and used for all exceptions that are not handled within the route structure itself.

The second case allows you to restrict the applicability of your handler to certain branches of your route structure.

Here is an example for wiring up a custom handler via *handleExceptions*:

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.http.scaladsl.server._
import StatusCodes._
import Directives._

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

object MyApp extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()

  val route: Route =
    handleExceptions(myExceptionHandler) {
      // ... some route structure
    }

  Http().bindAndHandle(route, "localhost", 8080)
}
```

And this is how to do it implicitly:

```
import akka.actor.ActorSystem
import akka.stream.ActorMaterializer
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.http.scaladsl.server._
import StatusCodes._
import Directives._

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

object MyApp extends App {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()

  val route: Route =
  // ... some route structure

  Http().bindAndHandle(route, "localhost", 8080)
}
```

2.5.6 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* or *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 (unsigned) 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 (unsigned) 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")
```

2.5.7 Case Class Extraction

The value extraction performed by *Directives* is a nice way of providing your route logic with interesting request properties, all with proper type-safety and error handling. However, in some case you might want even more. Consider this example:

```
case class Color(red: Int, green: Int, blue: Int)

val route =
  path("color") {
    parameters('red.as[Int], 'green.as[Int], 'blue.as[Int]) { (red, green, blue) =>
      val color = Color(red, green, blue)
```

```
      // ... route working with the `color` instance
    }
  }
```

Here the *parameters* directives is employed to extract three `Int` values, which are then used to construct an instance of the `Color` case class. So far so good. However, if the model classes we'd like to work with have more than just a few parameters the overhead introduced by capturing the arguments as extractions only to feed them into the model class constructor directly afterwards can somewhat clutter up your route definitions.

If your model classes are case classes, as in our example, Akka HTTP supports an even shorter and more concise syntax. You can also write the example above like this:

```
case class Color(red: Int, green: Int, blue: Int)

val route =
  path("color") {
    parameters('red.as[Int], 'green.as[Int], 'blue.as[Int]).as(Color) { color =>
      // ... route working with the `color` instance
    }
  }
```

You can postfix any directive with extractions with an `as(...)` call. By simply passing the companion object of your model case class to the `as` modifier method the underlying directive is transformed into an equivalent one, which extracts only one value of the type of your model class. Note that there is no reflection involved and your case class does not have to implement any special interfaces. The only requirement is that the directive you attach the `as` call to produces the right number of extractions, with the right types and in the right order.

If you'd like to construct a case class instance from extractions produced by *several* directives you can first join the directives with the `&` operator before using the `as` call:

```
case class Color(name: String, red: Int, green: Int, blue: Int)

val route =
  (path("color" / Segment) & parameters('r.as[Int], 'g.as[Int], 'b.as[Int]))
    .as(Color) { color =>
      // ... route working with the `color` instance
    }
```

Here the `Color` class has gotten another member, `name`, which is supplied not as a parameter but as a path element. By joining the `path` and `parameters` directives with `&` you create a directive extracting 4 values, which directly fit the member list of the `Color` case class. Therefore you can use the `as` modifier to convert the directive into one extracting only a single `Color` instance.

Generally, when you have routes that work with, say, more than 3 extractions it's a good idea to introduce a case class for these and resort to case class extraction. Especially since it supports another nice feature: validation.

Caution: There is one quirk to look out for when using case class extraction: If you create an explicit companion object for your case class, no matter whether you actually add any members to it or not, the syntax presented above will not (quite) work anymore. Instead of `as(Color)` you will then have to say `as(Color.apply)`. This behavior appears as if it's not really intended, so this might be improved in future Scala versions.

Case Class Validation

In many cases your web service needs to verify input parameters according to some logic before actually working with them. E.g. in the example above the restriction might be that all color component values must be between 0 and 255. You could get this done with a few *validate* directives but this would quickly become cumbersome and hard to read.

If you use case class extraction you can put the verification logic into the constructor of your case class, where it should be:

```
case class Color(name: String, red: Int, green: Int, blue: Int) {
  require(!name.isEmpty, "color name must not be empty")
  require(0 <= red && red <= 255, "red color component must be between 0 and 255")
  require(0 <= green && green <= 255, "green color component must be between 0 and 255")
  require(0 <= blue && blue <= 255, "blue color component must be between 0 and 255")
}
```

If you write your validations like this Akka HTTP's case class extraction logic will properly pick up all error messages and generate a `ValidationRejection` if something goes wrong. By default, `ValidationRejections` are converted into `400 Bad Request` error response by the default *RejectionHandler*, if no subsequent route successfully handles the request.

2.5.8 Route TestKit

One of Akka HTTP's design goals is good testability of the created services. For services built with the Routing DSL Akka HTTP provides a dedicated testkit that makes efficient testing of route logic easy and convenient. This "route test DSL" is made available with the *akka-http-testkit* module.

Usage

Here is an example of what a simple test with the routing testkit might look like (using the built-in support for scalatest):

```
import org.scalatest.{ Matchers, WordSpec }
import akka.http.scaladsl.model.StatusCodes
import akka.http.scaladsl.testkit.ScalatestRouteTest
import akka.http.scaladsl.server._
import Directives._

class FullTestKitExampleSpec extends WordSpec with Matchers with ScalatestRouteTest {

  val smallRoute =
    get {
      pathSingleSlash {
        complete {
          "Captain on the bridge!"
        }
      } ~
      path("ping") {
        complete("PONG!")
      }
    }

  "The service" should {

    "return a greeting for GET requests to the root path" in {
      // tests:
      Get() ~> smallRoute ~> check {
        responseAs[String] shouldEqual "Captain on the bridge!"
      }
    }

    "return a 'PONG!' response for GET requests to /ping" in {
      // tests:
      Get("/ping") ~> smallRoute ~> check {
        responseAs[String] shouldEqual "PONG!"
      }
    }

    "leave GET requests to other paths unhandled" in {
```

```
      // tests:
      Get("/kermit") ~> smallRoute ~> check {
        handled shouldBe false
      }
    }

    "return a MethodNotAllowed error for PUT requests to the root path" in {
      // tests:
      Put() ~> Route.seal(smallRoute) ~> check {
        status === StatusCodes.MethodNotAllowed
        responseAs[String] shouldEqual "HTTP method not allowed, supported methods: GET"
      }
    }
  }
}
```

The basic structure of a test built with the testkit is this (expression placeholder in all-caps):

```
REQUEST ~> ROUTE ~> check {
  ASSERTIONS
}
```

In this template *REQUEST* is an expression evaluating to an `HttpRequest` instance. In most cases your test will, in one way or another, extend from `RouteTest` which itself mixes in the `akka.http.scaladsl.client.RequestBuilding` trait, which gives you a concise and convenient way of constructing test requests. [12]

ROUTE is an expression evaluating to a *Route*. You can specify one inline or simply refer to the route structure defined in your service.

The final element of the `~>` chain is a `check` call, which takes a block of assertions as parameter. In this block you define your requirements onto the result produced by your route after having processed the given request. Typically you use one of the defined "inspectors" to retrieve a particular element of the routes response and express assertions against it using the test DSL provided by your test framework. For example, with scalatest, in order to verify that your route responds to the request with a status 200 response, you'd use the `status` inspector and express an assertion like this:

```
status shouldEqual 200
```

The following inspectors are defined:

[12] If the request URI is relative it will be made absolute using an implicitly available instance of `DefaultHostInfo` whose value is "http://example.com" by default. This mirrors the behavior of akka-http-core which always produces absolute URIs for incoming request based on the request URI and the `Host`-header of the request. You can customize this behavior by bringing a custom instance of `DefaultHostInfo` into scope.

Inspector	Description
`charset: HttpCharset`	Identical to `contentType.charset`
`chunks: Seq[HttpEntity.ChunkStreamPart]`	Returns the entity chunks produced by the route. If the entity is not chunked returns `Nil`.
`closingExtension: String`	Returns chunk extensions the route produced with its last response chunk. If the response entity is unchunked returns the empty string.
`contentType: ContentType`	Identical to `responseEntity.contentType`
`definedCharset: Option[HttpCharset]`	Identical to `contentType.definedCharset`
`entityAs[T :FromEntityUnmarshaller T`	Unmarshals the response entity using the in-scope `FromEntityUnmarshaller` for the given type. Any errors in the process trigger a test failure.
`handled: Boolean`	Indicates whether the route produced an `HttpResponse` for the request. If the route rejected the request `handled` evaluates to `false`.
`header(name: String): Option[HttpHeader]`	Returns the response header with the given name or `None` if no such header is present in the response.
`header[T <: HttpHeader]: Option[T]`	Identical to `response.header[T]`
`headers: Seq[HttpHeader]`	Identical to `response.headers`
`mediaType: MediaType`	Identical to `contentType.mediaType`
`rejection: Rejection`	The rejection produced by the route. If the route did not produce exactly one rejection a test failure is triggered.
`rejections: Seq[Rejection]`	The rejections produced by the route. If the route did not reject the request a test failure is triggered.
`response: HttpResponse`	The `HttpResponse` returned by the route. If the route did not return an `HttpResponse` instance (e.g. because it rejected the request) a test failure is triggered.
`responseAs[T: FromResponseUnmarshaller T`	Unmarshals the response entity using the in-scope `FromResponseUnmarshaller` for the given type. Any errors in the process trigger a test failure.
`responseEntity: HttpEntity`	Returns the response entity.
`status: StatusCode`	Identical to `response.status`
`trailer: Seq[HttpHeader]`	Returns the list of trailer headers the route produced with its last chunk. If the response entity is unchunked returns `Nil`.

Sealing Routes

The section above describes how to test a "regular" branch of your route structure, which reacts to incoming requests with HTTP response parts or rejections. Sometimes, however, you will want to verify that your service also translates *Rejections* to HTTP responses in the way you expect.

You do this by wrapping your route with the `akka.http.scaladsl.server.Route.seal`. The `seal` wrapper applies the logic of the in-scope *ExceptionHandler* and *RejectionHandler* to all exceptions and rejections coming back from the route, and translates them to the respective `HttpResponse`.

Examples

A great pool of examples are the tests for all the predefined directives in Akka HTTP. They can be found here.

2.5.9 Server-Side WebSocket Support

WebSocket is a protocol that provides a bi-directional channel between browser and webserver usually run over an upgraded HTTP(S) connection. Data is exchanged in messages whereby a message can either be binary data or unicode text.

Akka HTTP provides a stream-based implementation of the WebSocket protocol that hides the low-level details of the underlying binary framing wire-protocol and provides a simple API to implement services using WebSocket.

Model

The basic unit of data exchange in the WebSocket protocol is a message. A message can either be binary message, i.e. a sequence of octets or a text message, i.e. a sequence of unicode code points.

Akka HTTP provides a straight-forward model for this abstraction:

```
/**
 * The ADT for Websocket messages. A message can either be a binary or a text message.
 */
sealed trait Message

/**
 * A binary
 */
sealed trait TextMessage extends Message {
  /**
   * The contents of this message as a stream.
   */
  def textStream: Source[String, _]
}
sealed trait BinaryMessage extends Message {
  /**
   * The contents of this message as a stream.
   */
  def dataStream: Source[ByteString, _]
}
```

The data of a message is provided as a stream because WebSocket messages do not have a predefined size and could (in theory) be infinitely long. However, only one message can be open per direction of the WebSocket connection, so that many application level protocols will want to make use of the delineation into (small) messages to transport single application-level data units like "one event" or "one chat message".

Many messages are small enough to be sent or received in one go. As an opportunity for optimization, the model provides a `Strict` subclass for each kind of message which contains data as a strict, i.e. non-streamed, `ByteString` or `String`.

When receiving data from the network connection the WebSocket implementation tries to create a `Strict` message whenever possible, i.e. when the complete data was received in one chunk. However, the actual chunking of messages over a network connection and through the various streaming abstraction layers is not deterministic from the perspective of the application. Therefore, application code must be able to handle both streaming and strict messages and not expect certain messages to be strict. (Particularly, note that tests against `localhost` will behave differently than tests against remote peers where data is received over a physical network connection.)

For sending data, use `TextMessage.apply(text: String)` to create a `Strict` message which is often the natural choice when the complete message has already been assembled. Otherwise, use `TextMessage.apply(textStream: Source[String, Any])` to create a streaming message from an Akka Stream source.

Server API

The entrypoint for the Websocket API is the synthetic `UpgradeToWebsocket` header which is added to a request if Akka HTTP encounters a Websocket upgrade request.

The Websocket specification mandates that details of the Websocket connection are negotiated by placing special-purpose HTTP-headers into request and response of the HTTP upgrade. In Akka HTTP these HTTP-level details of the WebSocket handshake are hidden from the application and don't need to be managed manually.

Instead, the synthetic `UpgradeToWebsocket` represents a valid Websocket upgrade request. An application can detect a Websocket upgrade request by looking for the `UpgradeToWebsocket` header. It can choose to accept the upgrade and start a Websocket connection by responding to that request with an `HttpResponse` generated by one of the `UpgradeToWebsocket.handleMessagesWith` methods. In its most general form this method expects two arguments: first, a handler `Flow[Message, Message, Any]` that will be used to handle Websocket messages on this connection. Second, the application can optionally choose one of the proposed application-level sub-protocols by inspecting the values of `UpgradeToWebsocket.requestedProtocols` and pass the chosen protocol value to `handleMessages`.

Handling Messages

A message handler is expected to be implemented as a `Flow[Message, Message, Any]`. For typical request-response scenarios this fits very well and such a `Flow` can be constructed from a simple function by using `Flow[Message].map` or `Flow[Message].mapAsync`.

There are other use-cases, e.g. in a server-push model, where a server message is sent spontaneously, or in a true bi-directional scenario where input and output aren't logically connected. Providing the handler as a `Flow` in these cases may not fit. Another method, `UpgradeToWebsocket.handleMessagesWithSinkSource`, is provided which allows to pass an output-generating `Source[Message, Any]` and an input-receiving `Sink[Message, Any]` independently.

Note that a handler is required to consume the data stream of each message to make place for new messages. Otherwise, subsequent messages may be stuck and message traffic in this direction will stall.

Example

Let's look at an example.

Websocket requests come in like any other requests. In the example, requests to `/greeter` are expected to be Websocket requests:

```
val requestHandler: HttpRequest ⇒ HttpResponse = {
  case req @ HttpRequest(GET, Uri.Path("/greeter"), _, _, _) ⇒
    req.header[UpgradeToWebsocket] match {
      case Some(upgrade) ⇒ upgrade.handleMessages(greeterWebsocketService)
      case None          ⇒ HttpResponse(400, entity = "Not a valid websocket request!")
    }
  case _: HttpRequest ⇒ HttpResponse(404, entity = "Unknown resource!")
}
```

It uses pattern matching on the path and then inspects the request to query for the `UpgradeToWebsocket` header. If such a header is found, it is used to generate a response by passing a handler for Websocket messages to the `handleMessages` method. If no such header is found a "400 Bad Request" response is generated.

The passed handler expects text messages where each message is expected to contain (a person's) name and then responds with another text message that contains a greeting:

```
// The Greeter WebSocket Service expects a "name" per message and
// returns a greeting message for that name
val greeterWebsocketService =
  Flow[Message]
    .mapConcat {
      // we match but don't actually consume the text message here,
      // rather we simply stream it back as the tail of the response
      // this means we might start sending the response even before the
      // end of the incoming message has been received
```

```
      case tm: TextMessage ⇒ TextMessage(Source.single("Hello ") ++ tm.textStream) :: Nil
      case bm: BinaryMessage =>
        // ignore binary messages but drain content to avoid the stream being clogged
        bm.dataStream.runWith(Sink.ignore)
        Nil
    }
```

Routing support

The routing DSL provides the *handleWebsocketMessages* directive to install a WebSocket handler if the request was a WebSocket request. Otherwise, the directive rejects the request.

Here's the above simple request handler rewritten as a route:

```
def greeter: Flow[Message, Message, Any] =
  Flow[Message].mapConcat {
    case tm: TextMessage ⇒
      TextMessage(Source.single("Hello ") ++ tm.textStream ++ Source.single("!")) :: Nil
    case bm: BinaryMessage ⇒
      // ignore binary messages but drain content to avoid the stream being clogged
      bm.dataStream.runWith(Sink.ignore)
      Nil
  }
val websocketRoute =
  path("greeter") {
    handleWebsocketMessages(greeter)
  }

// tests:
// create a testing probe representing the client-side
val wsClient = WSProbe()

// WS creates a Websocket request for testing
WS("/greeter", wsClient.flow) ~> websocketRoute ~>
  check {
    // check response for WS Upgrade headers
    isWebsocketUpgrade shouldEqual true

    // manually run a WS conversation
    wsClient.sendMessage("Peter")
    wsClient.expectMessage("Hello Peter!")

    wsClient.sendMessage(BinaryMessage(ByteString("abcdef")))
    wsClient.expectNoMessage(100.millis)

    wsClient.sendMessage("John")
    wsClient.expectMessage("Hello John!")

    wsClient.sendCompletion()
    wsClient.expectCompletion()
  }
```

The example also includes code demonstrating the testkit support for Websocket services. It allows to create Websocket requests to run against a route using *WS* which can be used to provide a mock Websocket probe that allows manual testing of the Websocket handler's behavior if the request was accepted.

2.5.10 Minimal Example

This is a complete, very basic Akka HTTP application relying on the Routing DSL:

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.marshallers.xml.ScalaXmlSupport._
import akka.http.scaladsl.server.Directives._
import akka.stream.ActorMaterializer

object Main extends App {
  implicit val system = ActorSystem("my-system")
  implicit val materializer = ActorMaterializer()
  implicit val ec = system.dispatcher

  val route =
    path("hello") {
      get {
        complete {
          <h1>Say hello to akka-http</h1>
        }
      }
    }

  val bindingFuture = Http().bindAndHandle(route, "localhost", 8080)

  println(s"Server online at http://localhost:8080/\nPress RETURN to stop...")
  Console.readLine() // for the future transformations
  bindingFuture
    .flatMap(_.unbind()) // trigger unbinding from the port
    .onComplete(_ ⇒ system.shutdown()) // and shutdown when done
}
```

It starts an HTTP Server on localhost and replies to GET requests to `/hello` with a simple response.

2.5.11 Longer Example

The following is an Akka HTTP route definition that tries to show off a few features. The resulting service does not really do anything useful but its definition should give you a feel for what an actual API definition with the Routing DSL will look like:

```
import akka.actor.ActorRef
import akka.http.scaladsl.coding.Deflate
import akka.http.scaladsl.marshalling.ToResponseMarshaller
import akka.http.scaladsl.model.StatusCodes.MovedPermanently
import akka.http.scaladsl.server.Directives._
// TODO: these explicit imports are only needed in complex cases, like below; Also, not needed on
import akka.http.scaladsl.server.directives.ParameterDirectives.ParamMagnet
import akka.http.scaladsl.server.directives.FormFieldDirectives.FieldMagnet
import akka.http.scaladsl.unmarshalling.FromRequestUnmarshaller
import akka.pattern.ask
import akka.util.Timeout

// types used by the API routes
type Money = Double // only for demo purposes, don't try this at home!
type TransactionResult = String
case class User(name: String)
case class Order(email: String, amount: Money)
case class Update(order: Order)
case class OrderItem(i: Int, os: Option[String], s: String)

// marshalling would usually be derived automatically using libraries
implicit val orderUM: FromRequestUnmarshaller[Order] = ???
implicit val orderM: ToResponseMarshaller[Order] = ???
implicit val orderSeqM: ToResponseMarshaller[Seq[Order]] = ???
implicit val timeout: Timeout = ??? // for actor asks
implicit val ec: ExecutionContext = ???
```

```
implicit val mat: ActorMaterializer = ???
implicit val sys: ActorSystem = ???

// backend entry points
def myAuthenticator: Authenticator[User] = ???
def retrieveOrdersFromDB: Seq[Order] = ???
def myDbActor: ActorRef = ???
def processOrderRequest(id: Int, complete: Order => Unit): Unit = ???

val route = {
  path("orders") {
    authenticateBasic(realm = "admin area", myAuthenticator) { user =>
      get {
        encodeResponseWith(Deflate) {
          complete {
            // marshal custom object with in-scope marshaller
            retrieveOrdersFromDB
          }
        }
      } ~
      post {
        // decompress gzipped or deflated requests if required
        decodeRequest {
          // unmarshal with in-scope unmarshaller
          entity(as[Order]) { order =>
            complete {
              // ... write order to DB
              "Order received"
            }
          }
        }
      }
    }
  } ~
  // extract URI path element as Int
  pathPrefix("order" / IntNumber) { orderId =>
    pathEnd {
      (put | parameter('method ! "put")) {
        // form extraction from multipart or www-url-encoded forms
        formFields(('email, 'total.as[Money])).as(Order) { order =>
          complete {
            // complete with serialized Future result
            (myDbActor ? Update(order)).mapTo[TransactionResult]
          }
        }
      } ~
      get {
        // debugging helper
        logRequest("GET-ORDER") {
          // use in-scope marshaller to create completer function
          completeWith(instanceOf[Order]) { completer =>
            // custom
            processOrderRequest(orderId, completer)
          }
        }
      }
    } ~
    path("items") {
      get {
        // parameters to case class extraction
        parameters(('size.as[Int], 'color ?, 'dangerous ? "no"))
          .as(OrderItem) { orderItem =>
            // ... route using case class instance created from
```

```
            // required and optional query parameters
          }
      }
    }
  } ~
  pathPrefix("documentation") {
    // optionally compresses the response with Gzip or Deflate
    // if the client accepts compressed responses
    encodeResponse {
      // serve up static content from a JAR resource
      getFromResourceDirectory("docs")
    }
  } ~
  path("oldApi" / Rest) { pathRest =>
    redirect("http://oldapi.example.com/" + pathRest, MovedPermanently)
  }
}
```

2.5.12 Handling HTTP Server failures in the High-Level API

There are various situations when failure may occur while initialising or running an Akka HTTP server. Akka by default will log all these failures, however sometimes one may want to react to failures in addition to them just being logged, for example by shutting down the actor system, or notifying some external monitoring end-point explicitly.

Bind failures

For example the server might be unable to bind to the given port. For example when the port is already taken by another application, or if the port is privileged (i.e. only usable by `root`). In this case the "binding future" will fail immediatly, and we can react to if by listening on the Future's completion:

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.server.Directives._
import akka.stream.ActorMaterializer

implicit val system = ActorSystem()
implicit val materializer = ActorMaterializer()
implicit val ec = system.dispatcher

val handler = get {
  complete("Hello world!")
}

// let's say the OS won't allow us to bind to 80.
val (host, port) = ("localhost", 80)
val bindingFuture: Future[ServerBinding] =
  Http().bindAndHandle(handler, host, port)

bindingFuture onFailure {
  case ex: Exception =>
    log.error(ex, "Failed to bind to {}:{}!", host, port)
}
```

Note: For a more low-level overview of the kinds of failures that can happen and also more fine-grained control over them refer to the *Handling HTTP Server failures in the Low-Level API* documentation.

Failures and exceptions inside the Routing DSL

Exception handling within the Routing DSL is done by providing `ExceptionHandler` s which are documented in-depth in the *Exception Handling* section of the documtnation. You can use them to transform exceptions into `HttpResponse` s with apropriate error codes and human-readable failure descriptions.

File uploads

For high level directives to handle uploads see the *FileUploadDirectives*.

Handling a simple file upload from for example a browser form with a *file* input can be done by accepting a *Multipart.FormData* entity, note that the body parts are *Source* rather than all available right away, and so is the individual body part payload so you will need to consume those streams both for the file and for the form fields.

Here is a simple example which just dumps the uploaded file into a temporary file on disk, collects some form fields and saves an entry to a fictive database:

```
val uploadVideo =
  path("video") {
    entity(as[Multipart.FormData]) { formData =>

      // collect all parts of the multipart as it arrives into a map
      val allPartsF: Future[Map[String, Any]] = formData.parts.mapAsync[(String, Any)](1) {

        case b: BodyPart if b.name == "file" =>
          // stream into a file as the chunks of it arrives and return a future
          // file to where it got stored
          val file = File.createTempFile("upload", "tmp")
          b.entity.dataBytes.runWith(FileIO.toFile(file)).map(_ =>
            (b.name -> file))

        case b: BodyPart =>
          // collect form field values
          b.toStrict(2.seconds).map(strict =>
            (b.name -> strict.entity.data.utf8String))

      }.runFold(Map.empty[String, Any])((map, tuple) => map + tuple)

      val done = allPartsF.map { allParts =>
        // You would have some better validation/unmarshalling here
        db.create(Video(
          file = allParts("file").asInstanceOf[File],
          title = allParts("title").asInstanceOf[String],
          author = allParts("author").asInstanceOf[String]))
      }

      // when processing have finished create a response for the user
      onSuccess(allPartsF) { allParts =>
        complete {
          "ok!"
        }
      }
    }
  }
```

You can transform the uploaded files as they arrive rather than storing then in a temporary file as in the previous example. In this example we accept any number of `.csv` files, parse those into lines and split each line before we send it to an actor for further processing:

```
val splitLines = Framing.delimiter(ByteString("\n"), 256)

val csvUploads =
```

```
path("metadata" / LongNumber) { id =>
  entity(as[Multipart.FormData]) { formData =>
    val done = formData.parts.mapAsync(1) {
      case b: BodyPart if b.filename.exists(_.endsWith(".csv")) =>
        b.entity.dataBytes
          .via(splitLines)
          .map(_.utf8String.split(",").toVector)
          .runForeach(csv =>
            metadataActor ! MetadataActor.Entry(id, csv))
      case _ => Future.successful(Unit)
    }.runWith(Sink.ignore)

    // when processing have finished create a response for the user
    onSuccess(done) {
      complete {
        "ok!"
      }
    }
  }
}
```

2.6 Consuming HTTP-based Services (Client-Side)

All client-side functionality of Akka HTTP, for consuming HTTP-based services offered by other endpoints, is currently provided by the `akka-http-core` module.

Depending on your application's specific needs you can choose from three different API levels:

Connection-Level Client-Side API for full-control over when HTTP connections are opened/closed and how requests are scheduled across them

Host-Level Client-Side API for letting Akka HTTP manage a connection-pool to *one specific* host/port endpoint

Request-Level Client-Side API for letting Akka HTTP perform all connection management

You can interact with different API levels at the same time and, independently of which API level you choose, Akka HTTP will happily handle many thousand concurrent connections to a single or many different hosts.

2.6.1 Connection-Level Client-Side API

The connection-level API is the lowest-level client-side API Akka HTTP provides. It gives you full control over when HTTP connections are opened and closed and how requests are to be send across which connection. As such it offers the highest flexibility at the cost of providing the least convenience.

Opening HTTP Connections

With the connection-level API you open a new HTTP connection to a target endpoint by materializing a `Flow` returned by the `Http().outgoingConnection(...)` method. Here is an example:

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.stream.ActorMaterializer
import akka.stream.scaladsl._

import scala.concurrent.Future

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

```
val connectionFlow: Flow[HttpRequest, HttpResponse, Future[Http.OutgoingConnection]] =
  Http().outgoingConnection("akka.io")
val responseFuture: Future[HttpResponse] =
  Source.single(HttpRequest(uri = "/"))
    .via(connectionFlow)
    .runWith(Sink.head)
```

Apart from the host name and port the `Http().outgoingConnection(...)` method also allows you to specify socket options and a number of configuration settings for the connection.

Note that no connection is attempted until the returned flow is actually materialized! If the flow is materialized several times then several independent connections will be opened (one per materialization). If the connection attempt fails, for whatever reason, the materialized flow will be immediately terminated with a respective exception.

Request-Response Cycle

Once the connection flow has been materialized it is ready to consume `HttpRequest` instances from the source it is attached to. Each request is sent across the connection and incoming responses dispatched to the downstream pipeline. Of course and as always, back-pressure is adequately maintained across all parts of the connection. This means that, if the downstream pipeline consuming the HTTP responses is slow, the request source will eventually be slowed down in sending requests.

Any errors occurring on the underlying connection are surfaced as exceptions terminating the response stream (and canceling the request source).

Note that, if the source produces subsequent requests before the prior responses have arrived, these requests will be pipelined across the connection, which is something that is not supported by all HTTP servers. Also, if the server closes the connection before responses to all requests have been received this will result in the response stream being terminated with a truncation error.

Closing Connections

Akka HTTP actively closes an established connection upon reception of a response containing `Connection: close` header. The connection can also be closed by the server.

An application can actively trigger the closing of the connection by completing the request stream. In this case the underlying TCP connection will be closed when the last pending response has been received.

Timeouts

Currently Akka HTTP doesn't implement client-side request timeout checking itself as this functionality can be regarded as a more general purpose streaming infrastructure feature. However, akka-stream should soon provide such a feature.

Stand-Alone HTTP Layer Usage

Due to its Reactive-Streams-based nature the Akka HTTP layer is fully detachable from the underlying TCP interface. While in most applications this "feature" will not be crucial it can be useful in certain cases to be able to "run" the HTTP layer (and, potentially, higher-layers) against data that do not come from the network but rather some other source. Potential scenarios where this might be useful include tests, debugging or low-level event-sourcing (e.g by replaying network traffic).

On the client-side the stand-alone HTTP layer forms a `BidiStage` that is defined like this:

```
/**
 * The type of the client-side HTTP layer as a stand-alone BidiFlow
 * that can be put atop the TCP layer to form an HTTP client.
 *
 * {{{
```

```
 *                +------+
 * HttpRequest  ~>|      |~> SslTlsOutbound
 *                | bidi |
 * HttpResponse <~|      |<~ SslTlsInbound
 *                +------+
 * }}}
 */
type ClientLayer = BidiFlow[HttpRequest, SslTlsOutbound, SslTlsInbound, HttpResponse, Unit]
```

You create an instance of `Http.ClientLayer` by calling one of the two overloads of the `Http().clientLayer` method, which also allows for varying degrees of configuration.

2.6.2 Host-Level Client-Side API

As opposed to the *Connection-Level Client-Side API* the host-level API relieves you from manually managing individual HTTP connections. It autonomously manages a configurable pool of connections to *one particular target endpoint* (i.e. host/port combination).

Requesting a Host Connection Pool

The best way to get a hold of a connection pool to a given target endpoint is the `Http().cachedHostConnectionPool(...)` method, which returns a `Flow` that can be "baked" into an application-level stream setup. This flow is also called a "pool client flow".

The connection pool underlying a pool client flow is cached. For every `ActorSystem`, target endpoint and pool configuration there will never be more than a single pool live at any time.

Also, the HTTP layer transparently manages idle shutdown and restarting of connection pools as configured. The client flow instances therefore remain valid throughout the lifetime of the application, i.e. they can be materialized as often as required and the time between individual materialization is of no importance.

When you request a pool client flow with `Http().cachedHostConnectionPool(...)` Akka HTTP will immediately start the pool, even before the first client flow materialization. However, this running pool will not actually open the first connection to the target endpoint until the first request has arrived.

Configuring a Host Connection Pool

Apart from the connection-level config settings and socket options there are a number of settings that allow you to influence the behavior of the connection pool logic itself. Check out the `akka.http.client.host-connection-pool` section of the Akka HTTP *Configuration* for more information about which settings are available and what they mean.

Note that, if you request pools with different configurations for the same target host you will get *independent* pools. This means that, in total, your application might open more concurrent HTTP connections to the target endpoint than any of the individual pool's `max-connections` settings allow!

There is one setting that likely deserves a bit deeper explanation: `max-open-requests`. This setting limits the maximum number of requests that can be in-flight at any time for a single connection pool. If an application calls `Http().cachedHostConnectionPool(...)` 3 times (with the same endpoint and settings) it will get back `3` different client flow instances for the same pool. If each of these client flows is then materialized `4` times (concurrently) the application will have 12 concurrently running client flow materializations. All of these share the resources of the single pool.

This means that, if the pool's `pipelining-limit` is left at `1` (effecitvely disabeling pipelining), no more than 12 requests can be open at any time. With a `pipelining-limit` of `8` and 12 concurrent client flow materializations the theoretical open requests maximum is `96`.

The `max-open-requests` config setting allows for applying a hard limit which serves mainly as a protection against erroneous connection pool use, e.g. because the application is materializing too many client flows that all compete for the same pooled connections.

Using a Host Connection Pool

The "pool client flow" returned by `Http().cachedHostConnectionPool(...)` has the following type:

```
Flow[(HttpRequest, T), (Try[HttpResponse], T), HostConnectionPool]
```

This means it consumes tuples of type `(HttpRequest, T)` and produces tuples of type `(Try[HttpResponse], T)` which might appear more complicated than necessary on first sight. The reason why the pool API includes objects of custom type `T` on both ends lies in the fact that the underlying transport usually comprises more than a single connection and as such the pool client flow often generates responses in an order that doesn't directly match the consumed requests. We could have built the pool logic in a way that reorders responses according to their requests before dispatching them to the application, but this would have meant that a single slow response could block the delivery of potentially many responses that would otherwise be ready for consumption by the application.

In order to prevent unnecessary head-of-line blocking the pool client-flow is allowed to dispatch responses as soon as they arrive, independently of the request order. Of course this means that there needs to be another way to associate a response with its respective request. The way that this is done is by allowing the application to pass along a custom "context" object with the request, which is then passed back to the application with the respective response. This context object of type `T` is completely opaque to Akka HTTP, i.e. you can pick whatever works best for your particular application scenario.

Note: A consequence of using a pool is that long-running requests block a connection while running and may starve other requests. Make sure not to use a connection pool for long-running requests like long-polling GET requests. Use the *Connection-Level Client-Side API* instead.

Connection Allocation Logic

This is how Akka HTTP allocates incoming requests to the available connection "slots":

1. If there is a connection alive and currently idle then schedule the request across this connection.
2. If no connection is idle and there is still an unconnected slot then establish a new connection.
3. If all connections are already established and "loaded" with other requests then pick the connection with the least open requests (< the configured `pipelining-limit`) that only has requests with idempotent methods scheduled to it, if there is one.
4. Otherwise apply back-pressure to the request source, i.e. stop accepting new requests.

For more information about scheduling more than one request at a time across a single connection see this wikipedia entry on HTTP pipelining.

Retrying a Request

If the `max-retries` pool config setting is greater than zero the pool retries idempotent requests for which a response could not be successfully retrieved. Idempotent requests are those whose HTTP method is defined to be idempotent by the HTTP spec, which are all the ones currently modelled by Akka HTTP except for the `POST`, `PATCH` and `CONNECT` methods.

When a response could not be received for a certain request there are essentially three possible error scenarios:

1. The request got lost on the way to the server.
2. The server experiences a problem while processing the request.
3. The response from the server got lost on the way back.

Since the host connector cannot know which one of these possible reasons caused the problem and therefore `PATCH` and `POST` requests could have already triggered a non-idempotent action on the server these requests cannot be retried.

In these cases, as well as when all retries have not yielded a proper response, the pool produces a failed `Try` (i.e. a `scala.util.Failure`) together with the custom request context.

Pool Shutdown

Completing a pool client flow will simply detach the flow from the pool. The connection pool itself will continue to run as it may be serving other client flows concurrently or in the future. Only after the configured `idle-timeout` for the pool has expired will Akka HTTP automatically terminate the pool and free all its resources.

If a new client flow is requested with `Http().cachedHostConnectionPool(...)` or if an already existing client flow is re-materialized the respective pool is automatically and transparently restarted.

In addition to the automatic shutdown via the configured idle timeouts it's also possible to trigger the immediate shutdown of a specific pool by calling `shutdown()` on the `HostConnectionPool` instance that the pool client flow materializes into. This `shutdown()` call produces a `Future[Unit]` which is fulfilled when the pool termination has been completed.

It's also possible to trigger the immediate termination of *all* connection pools in the `ActorSystem` at the same time by calling `Http().shutdownAllConnectionPools()`. This call too produces a `Future[Unit]` which is fulfilled when all pools have terminated.

Example

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.stream.ActorMaterializer
import akka.stream.scaladsl._

import scala.concurrent.Future
import scala.util.Try

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

// construct a pool client flow with context type `Int`
val poolClientFlow = Http().cachedHostConnectionPool[Int]("akka.io")
val responseFuture: Future[(Try[HttpResponse], Int)] =
  Source.single(HttpRequest(uri = "/") -> 42)
    .via(poolClientFlow)
    .runWith(Sink.head)
```

2.6.3 Request-Level Client-Side API

The request-level API is the most convenient way of using Akka HTTP's client-side functionality. It internally builds upon the *Host-Level Client-Side API* to provide you with a simple and easy-to-use way of retrieving HTTP responses from remote servers. Depending on your preference you can pick the flow-based or the future-based variant.

Note: The request-level API is implemented on top of a connection pool that is shared inside the ActorSystem. A consequence of using a pool is that long-running requests block a connection while running and starve other requests. Make sure not to use the request-level API for long-running requests like long-polling GET requests. Use the *Connection-Level Client-Side API* instead.

Flow-Based Variant

The flow-based variant of the request-level client-side API is presented by the `Http().superPool(...)` method. It creates a new "super connection pool flow", which routes incoming requests to a (cached) host connection pool depending on their respective effective URIs.

The `Flow` returned by `Http().superPool(...)` is very similar to the one from the *Host-Level Client-Side API*, so the *Using a Host Connection Pool* section also applies here.

However, there is one notable difference between a "host connection pool client flow" for the host-level API and a "super-pool flow": Since in the former case the flow has an implicit target host context the requests it takes don't need to have absolute URIs or a valid `Host` header. The host connection pool will automatically add a `Host` header if required.

For a super-pool flow this is not the case. All requests to a super-pool must either have an absolute URI or a valid `Host` header, because otherwise it'd be impossible to find out which target endpoint to direct the request to.

Future-Based Variant

Sometimes your HTTP client needs are very basic. You simply need the HTTP response for a certain request and don't want to bother with setting up a full-blown streaming infrastructure.

For these cases Akka HTTP offers the `Http().singleRequest(...)` method, which simply turns an `HttpRequest` instance into `Future[HttpResponse]`. Internally the request is dispatched across the (cached) host connection pool for the request's effective URI.

Just like in the case of the super-pool flow described above the request must have either an absolute URI or a valid `Host` header, otherwise the returned future will be completed with an error.

Using the Future-Based API in Actors

When using the `Future` based API from inside an `Actor`, all the usual caveats apply to how one should deal with the futures completion. For example you should not access the Actors state from within the Future's callbacks (such as `map`, `onComplete`, ...) and instead you should use the `pipeTo` pattern to pipe the result back to the Actor as a message.

```
import akka.actor.Actor
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.stream.scaladsl.ImplicitMaterializer

class Myself extends Actor
  with ImplicitMaterializer
  with ActorLogging {

  import akka.pattern.pipe
  import context.dispatcher

  val http = Http(context.system)

  override def preStart() = {
    http.singleRequest(HttpRequest(uri = "http://akka.io"))
      .pipeTo(self)
  }

  def receive = {
    case HttpResponse(StatusCodes.OK, headers, entity, _) =>
      log.info("Got response, body: " + entity.dataBytes.runFold(ByteString(""))(_ ++ _))
    case HttpResponse(code, _, _, _) =>
      log.info("Request failed, response code: " + code)
  }
```

```
}
```

An `ActorMaterializer` instance needed for Http to perfom its duties can be obtained using the `ImplicitMaterializer` helper trait.

Example

```
import akka.http.scaladsl.Http
import akka.http.scaladsl.model._
import akka.stream.ActorMaterializer

import scala.concurrent.Future

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

val responseFuture: Future[HttpResponse] =
  Http().singleRequest(HttpRequest(uri = "http://akka.io"))
```

Warning: Be sure to consume the response entities `dataBytes:Source[ByteString,Unit]` by for example connecting it to a `Sink` (for example `response.entity.dataBytes.runWith(Sink.ignore)` if you don't care about the response entity), since otherwise Akka HTTP (and the underlying Streams infrastructure) will understand the lack of entity consumption as a back-pressure signal and stop reading from the underlying TCP connection!
This is a feature of Akka HTTP that allows consuming entities (and pulling them through the network) in a streaming fashion, and only *on demand* when the client is ready to consume the bytes - it may be a bit suprising at first though.
There are tickets open about automatically dropping entities if not consumed (#18716 and #18540), so these may be implemented in the near future.

2.6.4 Client-Side HTTPS Support

Akka HTTP supports TLS encryption on the client-side as well as on the *server-side*.

The central vehicle for configuring encryption is the `HttpsContext`, which is defined as such:

```
final case class HttpsContext(sslContext: SSLContext,
                              enabledCipherSuites: Option[immutable.Seq[String]] = None,
                              enabledProtocols: Option[immutable.Seq[String]] = None,
                              clientAuth: Option[ClientAuth] = None,
                              sslParameters: Option[SSLParameters] = None)
```

In addition to the `outgoingConnection`, `newHostConnectionPool` and `cachedHostConnectionPool` methods the akka.http.scaladsl.Http extension also defines `outgoingConnectionTls`, `newHostConnectionPoolTls` and `cachedHostConnectionPoolTls`. These methods work identically to their counterparts without the `-Tls` suffix, with the exception that all connections will always be encrypted.

The `singleRequest` and `superPool` methods determine the encryption state via the scheme of the incoming request, i.e. requests to an "https" URI will be encrypted, while requests to an "http" URI won't.

The encryption configuration for all HTTPS connections, i.e. the `HttpsContext` is determined according to the following logic:

1. If the optional `httpsContext` method parameter is defined it contains the configuration to be used (and thus takes precedence over any potentially set default client-side `HttpsContext`).

2. If the optional `httpsContext` method parameter is undefined (which is the default) the default client-side `HttpsContext` is used, which can be set via the `setDefaultClientHttpsContext` on the `Http` extension.
3. If no default client-side `HttpsContext` has been set via the `setDefaultClientHttpsContext` on the `Http` extension the default system configuration is used.

Usually the process is, if the default system TLS configuration is not good enough for your application's needs, that you configure a custom `HttpsContext` instance and set it via `Http().setDefaultClientHttpsContext`. Afterwards you simply use `outgoingConnectionTls`, `newHostConnectionPoolTls`, `cachedHostConnectionPoolTls`, `superPool` or `singleRequest` without a specific `httpsContext` argument, which causes encrypted connections to rely on the configured default client-side `HttpsContext`.

If no custom `HttpsContext` is defined the default context uses Java's default TLS settings. Customizing the `HttpsContext` can make the Https client less secure. Understand what you are doing!

Hostname verification

Hostname verification proves that the Akka HTTP client is actually communicating with the server it intended to communicate with. Without this check a man-in-the-middle attack is possible. In the attack scenario, an alternative certificate would be presented which was issued for another host name. Checking the host name in the certificate against the host name the connection was opened against is therefore vital.

The default `HttpsContext` enables hostname verification. Akka HTTP relies on the Typesafe SSL-Config library to implement this and security options for SSL/TLS. Hostname verification is provided by the JDK and used by Akka HTTP since Java 7, and on Java 6 the verification is implemented by ssl-config manually.

Note: We highly recommend updating your Java runtime to the latest available release, preferably JDK 8, as it includes this and many more security features related to TLS.

2.6.5 Client-Side WebSocket Support

Not yet implemented see 17275.

2.7 Migration Guide from spray

TODO - will be written shortly.

- `respondWithStatus` also known as `overrideStatusCode` has not been forward ported to Akka HTTP, as it has been seen mostly as an anti-pattern. More information here: https://github.com/akka/akka/issues/18626
- `respondWithMediaType` was considered an anti-pattern in spray and is not ported to Akka HTTP. Instead users should rely on content type negotiation as Akka HTTP implements it. More information here: https://github.com/akka/akka/issues/18625