(https://www.lightbend.com)

Akka Documentation (../index.html)

Version 2.7.0

Scala

Search

Security Announcements (../security/inde...

Getting Started Guide (../typed/guide/ind...

General Concepts (../general/index.html)

Actors (../typed/index.html)

Cluster (../typed/index-cluster.html)

Persistence (Event Sourcing) (../typed/ind...

Persistence (Durable State) (../typed/inde...

Streams (../stream/index.html)

- Module info (../stream/index.html#module...
- **Introduction (../stream/stream-introduct...**
- **Streams Quickstart Guide (../stream/str...**
- **Design Principles behind Akka Streams ...**
- **Basics and working with Flows (../strea...**
- **Working with Graphs (../stream/stream-...**
- **Modularity, Composition and Hierarchy ...**

(https://akka.io)

Buffers and working with rate (../stream...

Context Propagation (../stream/stream-...

Dynamic stream handling (../stream/stre...

Custom stream processing (../stream/str...

Futures interop (../stream/futures-intero...

Actors interop (../stream/actor-interop.h...

Reactive Streams Interop (../stream/rea...

Error Handling in Streams (../stream/str...

Working with streaming IO (../stream/str...

StreamRefs - Reactive Streams over the ...

Pipelining and Parallelism (../stream/str...

Testing streams

Dependency

Introduction

Built-in sources, sinks and operators

TestKit

Streams TestKit

Fuzzing Mode

Substreams (../stream/stream-substrea...

Streams Cookbook (../stream/stream-co...

Configuration (../general/stream/stream...

Operators (../stream/operators/index.ht...

Discovery (../discovery/index.html)

Utilities (../index-utilities.html)

(https://akka.io)

Other Akka modules (../common/other-m...

Package, Deploy and Run (../additional/d...

Project Information (../project/index.html)

Akka Classic (../index-classic.html)

Testing streams

Dependency

To use Akka Stream TestKit, add the module to your project:

sbt **Maven** Gradle

```
<properties>
  <scala.binary.version>2.13</scala.binary.version>
</properties>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.typesafe.akka</groupId>
      <artifactId>akka-bom_${scala.binary.version}</artifactId>
      <version>2.7.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
```

```
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-stream-testkit_${scala.binary.version}</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Introduction

Verifying behavior 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` **(https://doc.akka.io/api/akka/2.7/akka/testkit/TestProbe.html)** 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 testable by wiring them up to other sources or sinks, or some test harnesses that `akka-testkit` or `akka-stream-testkit` provide.

Built-in sources, sinks and operators

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:

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L21-L26) copy

```
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, 3.seconds)
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 **`take`** **(https://doc.akka.io/api/akka/2.7/akka/stream/scaladsl/Source.html#take(n:Long):FlowOps.this.Repr[Out])** operator and **`Sink.seq`** **(https://doc.akka.io/api/akka/2.7/akka/stream/scaladsl/Sink$.html#seq[T]:akka.stream.scaladsl.Sink[T,scala.concurrent.Future[Seq[T]]])** are very useful.

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L32-L39) copy

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

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

val future = sourceUnderTest.take(10).runWith(Sink.seq)
val result = Await.result(future, 3.seconds)
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.

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L45-L49) copy

```
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, 3.seconds)
assert(result == (1 to 4))
```

TestKit

Akka Stream offers integration with Actors out of the box. This support can be used for writing stream tests that use familiar `TestProbe` **(https://doc.akka.io/api/akka/2.7/akka/testkit/TestProbe.html)** from the `akka-testkit` API.

One of the more straightforward tests would be to materialize stream to a `Future` **(https://www.scala-lang.org/api/2.13.10/scala/concurrent/Future.html)** and then use `pipe` **(https://doc.akka.io/api/akka/2.7/akka/pattern/PipeToSupport.html#pipe[T](future:scala.concurrent.Future[T])(implicitexecutionContext:scala.concurrent.ExecutionContext):PipeToSupport.this.PipeableFuture[T])** pattern to pipe the result of that future to the probe.

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L55-L62) copy

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

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

val probe = TestProbe()
sourceUnderTest.runWith(Sink.seq).pipeTo(probe.ref)
probe.expectMsg(3.seconds, Seq(Seq(1, 2), Seq(3, 4)))
```

Instead of materializing to a future, we can use a `Sink.actorRef` **(https://doc.akka.io/api/akka/2.7/akka/stream/scaladsl/Sink$.html#actorRef[T](ref:akka.actor.ActorRef,onCompleteMessage:Any,onFailureMessage:Throwable=%3EAny):akka.stream.scaladsl.Sink[T,akka.NotUsed])** that sends all incoming elements to the given `ActorRef` **(https://doc.akka.io/api/akka/2.7/akka/actor/ActorRef.html)**. Now we can use assertion methods on `TestProbe` **(https://doc.akka.io/api/akka/2.7/akka/testkit/TestProbe.html)** 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`.

(https://akka.io)

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L68-L80) copy

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

val probe = TestProbe()
val cancellable = sourceUnderTest
  .to(Sink.actorRef(probe.ref, onCompleteMessage = "completed", onFailureMessage = _ => "failed"))
  .run()

probe.expectMsg(1.second, Tick)
probe.expectNoMessage(100.millis)
probe.expectMsg(3.seconds, Tick)
cancellable.cancel()
probe.expectMsg(3.seconds, "completed")
```

Similarly to `Sink.actorRef` that provides control over received elements, we can use [`Source.actorRef` (https://doc.akka.io/api/akka/2.7/akka/stream/scaladsl/Source$.html#actorRef[T](completionMatcher:PartialFunction[Any,akka.stream.CompletionStrategy],failureMatcher:PartialFunction[Any,Throwable],bufferSize:Int,o]() and have full control over elements to be sent.

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L86-L107) copy

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

val (ref, future) = Source
  .actorRef(
    completionMatcher = {
      case Done =>
        CompletionStrategy.draining
    },
    // Never fail the stream because of a message:
    failureMatcher = PartialFunction.empty,
    bufferSize = 8,
    overflowStrategy = OverflowStrategy.fail)
  .toMat(sinkUnderTest)(Keep.both)
  .run()

ref ! 1
ref ! 2
ref ! 3
ref ! Done

val result = Await.result(future, 3.seconds)
assert(result == "123")
```

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` **(https://doc.akka.io/api/akka/2.7/akka/stream/testkit/scaladsl/TestSource$.html)** and `TestSink` **(https://doc.akka.io/api/akka/2.7/akka/stream/testkit/scaladsl/TestSink$.html)** which provide sources and sinks that materialize to probes that allow fluent API.

Using the TestKit

A sink returned by `TestSink.probe` **(https://doc.akka.io/api/akka/2.7/akka/stream/testkit/scaladsl/TestSink$.html#probe[T](implicitsystem:akka.actor.ActorSystem):akka.stream.scaladsl.Sink[T,akka.stream.testkit.TestSubscriber.Probe[T]])** allows manual control over demand and assertions over elements coming downstream.

Scala **(https://akka.io)** Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L113-L115) copy

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

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

A source returned by `TestSource.probe` **(https://doc.akka.io/api/akka/2.7/akka/stream/testkit/scaladsl/TestSource$.html#probe[T](implicitsystem:akka.actor.ActorSystem):akka.stream.scaladsl.Source[T,akka.stream.testkit.TestPublisher.Probe[T]])** can be used for asserting demand or controlling when stream is completed or ended with an error.

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L121-L123) copy

```
val sinkUnderTest = Sink.cancelled

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

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

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L129-L134) copy

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

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

assert(future.failed.futureValue.getMessage == "boom")
```

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

(https://akka.io)

Scala Java

source (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/test/scala/docs/stream/StreamTestKitDocSpec.scala#L141-L155) copy

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

val (pub, sub) = TestSource[Int]().via(flowUnderTest).toMat(TestSink[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"))
```

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.

Pipelining and Parallelism (../stream/stream-parallelism.html)

Substreams (../stream/stream-substream.html)

Found an error in this documentation? The source code for this page can be found **here (https://github.com/akka/akka/tree/v2.7.0/akka-docs/src/main/paradox/stream/stream-testkit.md)**. Please feel free to edit and contribute a pull request.

(https://akka.io)