CODE.STAR

Akka HTTP course

Pim Verkerk

www.codestar.nl

CODE.STAR

Schedule

- TBD

Akka HTTP course

Part 1

Getting Started

CODE.STAR

History

http://spray.io

Spray

spray is an open-source toolkit for building **REST/HTTP**-based integration layers on top of **Scala** and **Akka**.

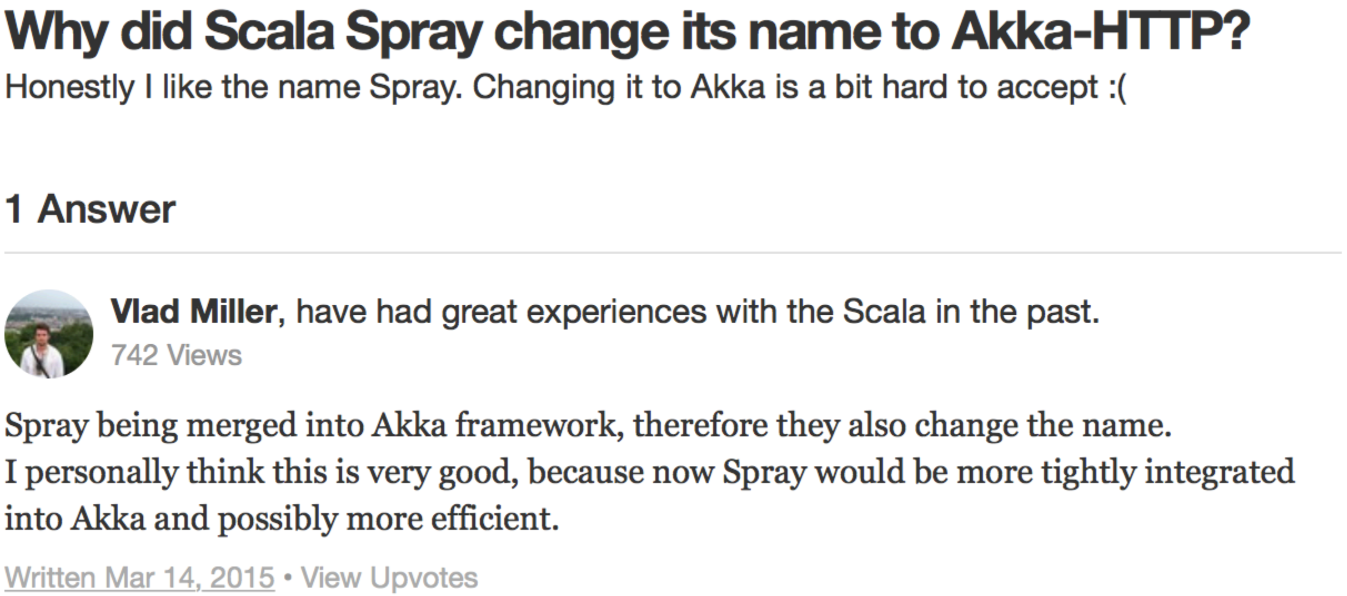
Being asynchronous, actor-based, fast, lightweight, modular and testable it's a great way to connect your **Scala** applications to the world.

Spray

- Elegant, high-performance HTTP for your Akka Actors
- Fast, lightweight HTTP Server
- Elegant DSL for API Construction
- Support for Servlet 3.0 Containers

Akka-HTTP

- Acquired by Lightbend
- ‘Spray 2.0’



Why did Scala Spray change its name to Akka-HTTP?

Honestly I like the name Spray. Changing it to Akka is a bit hard to accept :(

1 Answer

Vlad Miller, have had great experiences with the Scala in the past.

742 Views

Spray being merged into Akka framework, therefore they also change the name.
I personally think this is very good, because now Spray would be more tightly integrated into Akka and possibly more efficient.

Written Mar 14, 2015 • View Upvotes

CODE.STAR

REST

Akka HTTP

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 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~~

Threading model

- Thread per core
- Don’t block

HOW
DOES
IT
WORK

Getting started

```
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.

Getting started

```
// 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")
```

Getting started

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

Complex request

```
// 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`)
```

Getting started

```
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))
```

Main

```scala
import akka.http.scaladsl.server.Directives._
import …

object Main extends App {
  implicit val system = ActorSystem("hello-api")
  implicit val executor = system.dispatcher
  implicit val timeout = Timeout(1000.millis)

  implicit val materializer = ActorMaterializer()
  val serverBinding = Http().bindAndHandle(interface = "0.0.0.0", port = 8080, handler = mainFlow)

  def mainFlow(implicit system: ActorSystem, timeout: Timeout, executor: ExecutionContext): Route = {
    get {
      complete {
        "Hello World!"
      }
    }
  }
}
```

CODE.STAR

DSL / Directives

- get { … }
- complete {
- path("orders")
- post
- entity(as[Order]) { order =>
- get { … } ~ post { … }
- …

D Do
I It
Y Yourself

Exercise one

Clone https://github.com/code-star/akka-http-hello-world

1. Make a “Hello world” on a ‘get’
2. Add a page saying “Not so much hello?” on /noHello
3. Try to do a ‘post’ to /noHello
4. Make ‘post’ work also
5. Extra: Check if the header ‘IsCool’ is ‘true’

- Tip:

  https://chrome.google.com/webstore/detail/postman/

Akka HTTP course

Part 2

Build in Features

CODE.STAR

Composing Routes

Sub-Optimal

```
get {
    headerValueByName("IsCool") {
      case "true" => complete { "Your request is cool!" }
      case _      => reject
    } ~
      complete {
        "Hello World!"
      }
 }
```

Better!

```
def getRoute: Route = get { checkCool ~ notCool }

def checkCool: Route =
  headerValueByName("IsCool") {
    case "true" => complete { "Your request is cool!" }
    case _      => reject
  }

def notCool: Route =
  complete {
    "Hello World!"
  }
```

CONCURRENCY
Y U SO HARD?
DIYLOL.COM

Easy Future’s

```
Die.roll gives a Future[Int]

get {
   onSuccess(Die.roll) { roll =>
     complete { s"You rolled a $roll" }
   }
 }
```

Async

We have native support for concurrency with Future[ToResponseMarshallable]

This means we should have an implicit conversion from your response class to a akka-http response.

A String is ToResponseMarshallable so we use this in the next example

CODE.STAR

Complete with a Future[Reponse]

```
Introducing => ctx

   get { ctx =>
      val dieRoll: Future[Int] = Die.roll
      ctx.complete(
        dieRoll.map(roll => s"You rolled a $roll")
      )
    }
```

JSON support

```scala
import spray.json.DefaultJsonProtocol

case class Die(sides: Int)
case class Roll(die: Die, result: Int)

object RollProtocol extends DefaultJsonProtocol {
  implicit val fmtDie =  jsonFormat1(Die.apply)
  implicit val fmtRoll = jsonFormat2(Roll.apply)
}
```

JSON support

```
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
import RollProtocol._

 get {
      onSuccess(Roller.roll(Die(6))) { roll =>
        complete {
          roll
        }
      }
    }
```

```
roll is implicitly converted
```

```
Result:
{
  "die": {
    "sides": 6
  },
  "result": 6
}
```

Circuit breaker Pattern

Provides circuit breaker functionality to provide stability when working with "dangerous" operations, e.g. calls to remote systems

Or die rollers which are a bit sluggish ;)

Circuit breaker

```
val guard = CircuitBreaker (system.scheduler, 1, 1.second, 5.second)

 path("slow") {
      get {
        complete {
          guard.withCircuitBreaker(Roller.rollSlow(Die(6)))
        }
      }
    }
```

Error Handling

```
path("slow") {
      get {
        val guarded = guard.withCircuitBreaker(Roller.rollSlow(Die(6)))
        onComplete(guarded) {
          case Success(roll) => complete(roll)
          case _             => complete("Server Busy")
        }
      }
    }
```

D
Do
I
It
Y
Yourself

Exercise two

Checkout branch ‘exercise-two’

1. Give a Json response
2. Use the Circuit Breaker
3. Add error handling
4. Can you make a ‘loaded die’ behind a secret call ;)

- Tip:

http://doc.akka.io/docs/akka-stream-and-http-experimental/2.0.3/scala/http/index.html

Akka HTTP course

Part 3

WebSockets and Testing

CODE.STAR

Reactive Streams

- **Source**

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

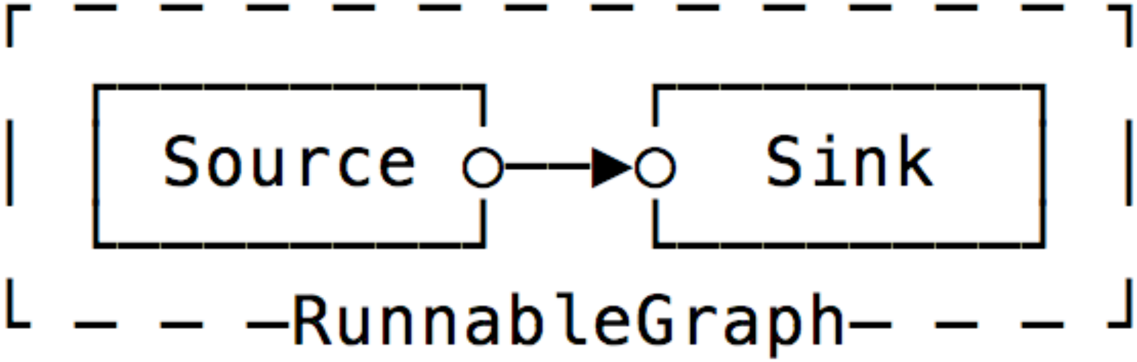
- **Sink**

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

- **Flow**

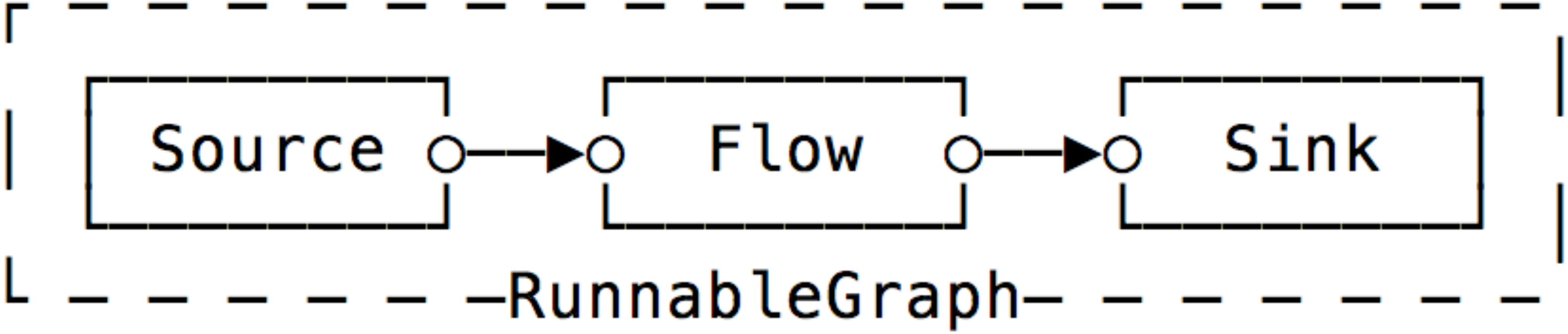
A processing stage which has exactly one input and output, which connects its up- and downstreams by transforming the data elements flowing through it.

In a picture



Source → Sink

RunnableGraph



Source → Flow → Sink

RunnableGraph

Running a Stream

- **RunnableGraph**

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

```scala
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()
```

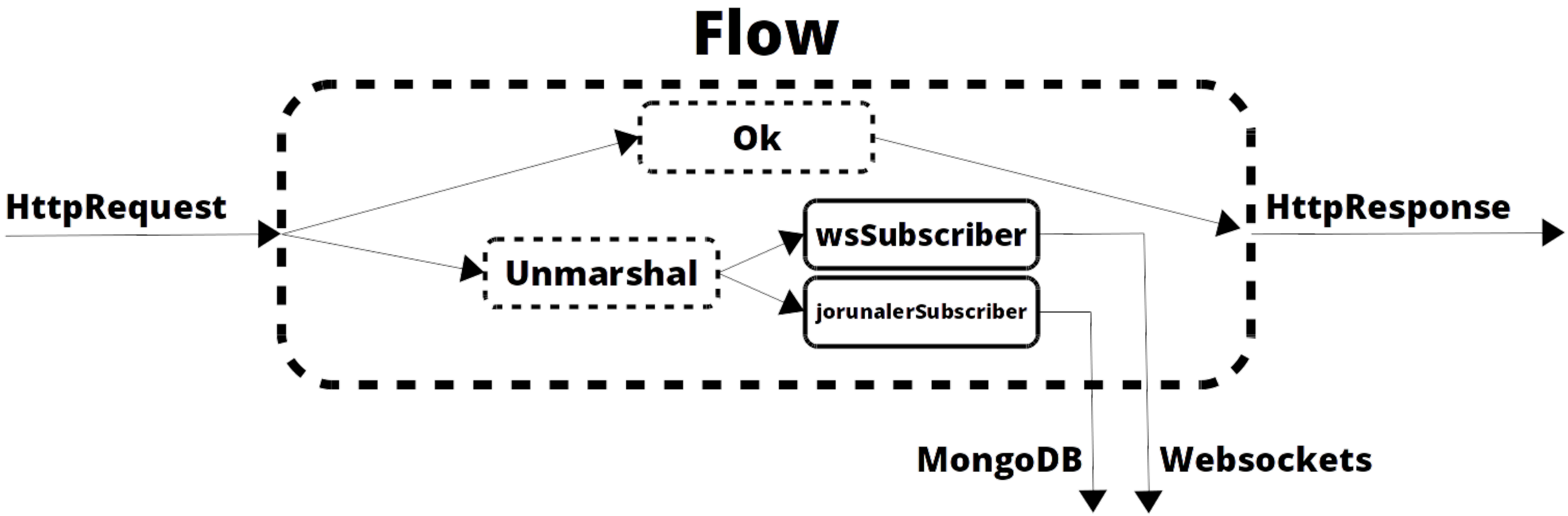
Async

Streams can create Future’s to

```
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)
```

Websockets


Flow
HttpRequest
Ok
Unmarshal
wsSubscriber
jorunalerSubscriber
HttpResponse
MongoDB
Websockets

Websockets

Create a websocket binding

```
 def allTweetsSocket = path("all") {
   handleWebSocketMessages(tweetFlow)
 }
```

Create a flow for a websocket

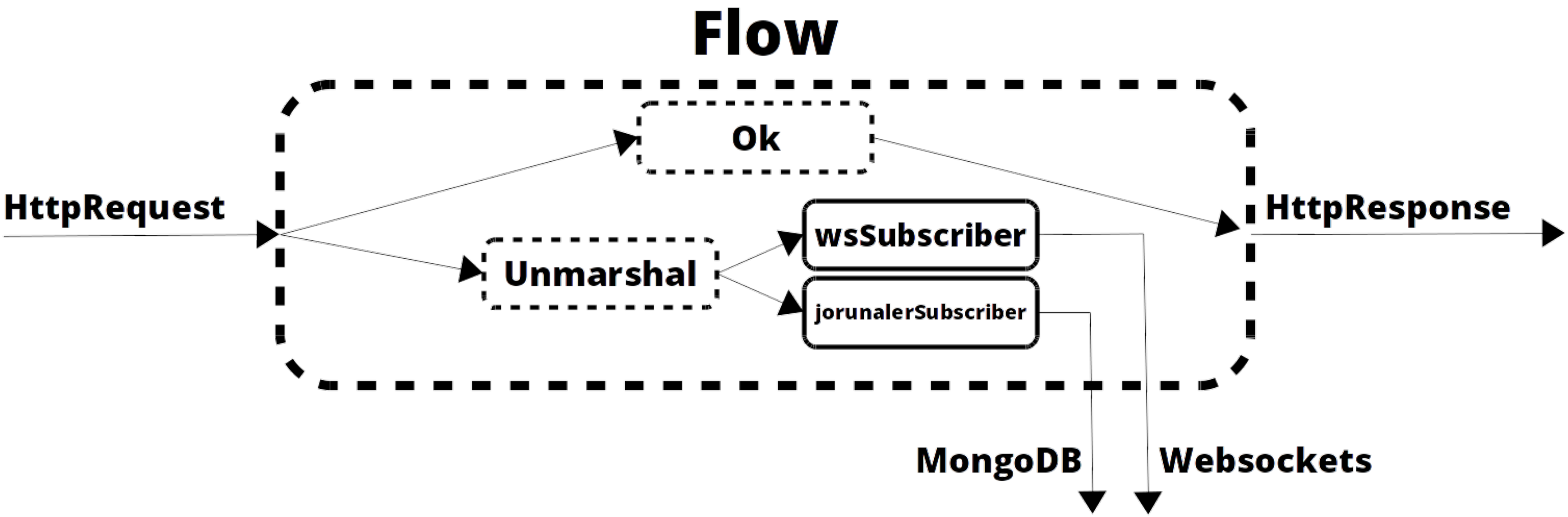
```
 private def tweetFlow: Flow[Message, Message, Unit] =
    Flow.fromSinkAndSource(Sink.ignore, tweetSource map toMessage)
```

The source

```
private val tweetSource: Source[Tweet, ActorRef] =
Source.actorPublisher[Tweet](TweetPublisher.props)
```

Sink.ignore?

Websockets


Flow
HttpRequest
Ok
Unmarshal
wsSubscriber
jorunalerSubscriber
HttpResponse
MongoDB
Websockets

Testing

```
class MainRoutingSpec extends FlatSpec with Matchers with
ScalatestRouteTest with TweetJsonProtocol {

  implicit val timeout = Timeout(1000.millis)
  val tweetActorManager = system.actorOf(TweetActorManager.props)

  "Main" should "serve the index page on /" in {
    Get("/") ~> Main.mainFlow ~> check {
      status shouldBe OK
    }
  }
}
```

Check REST status code

```
it should "allow to post a tweet for a user" in {
    Post("/resources/tweets", Tweet(User("test"), "Some tweet")) ~> Main.mainFlow ~> check {
        status shouldBe NoContent
    }
}
```

Check result entity

```
it should "serve tweets of a user on /resources/tweets/users/test" in {
   Get("/resources/tweets/users/test") ~> Main.mainFlow ~> check {
     status shouldBe OK
     contentType shouldBe `application/json`
     entityAs[String] should include regex ("Some tweet")
   }
 }
```

Check WebSocket responses

```
 it should "send tweets to the all websocket" in {
    val wsClient = WSProbe()

    WS("http://localhost/ws/tweets/all", wsClient.flow) ~> Main.mainFlow ~>
      check {
        isWebSocketUpgrade shouldEqual true

        tweetActorManager ! Tweet(User("test"), "Hello World!")
        wsClient.expectMessage("""{"user":{"name":"test"},"text":"Hello
World!"}""")
      }
  }
```

D Do
I It
Y Yourself

Exercise three

Clone https://github.com/J-Technologies/akka-http-websocket-activator-template.git

1. Added the template to Activator
    1. https://www.lightbend.com/activator/download
2. Follow the tutorial
3. Bonus: Look at the Low level API
    1. http://doc.akka.io/docs/akka-stream-and-http-experimental/2.0.3/scala/http/low-level-server-side-api.html
    2. Redo exercise-one in the low level API

Note: You might find some actors already. If you don’t understand them immediately, no worries; we will explain them tomorrow!

Wrap Up

Did you notice akka-http almost always find the correct http status en response type?

Akka-http does not force a single way to do things, you can choose from a lot of patterns

The testing DSL is very cool

Any comments?