REACTIVE APIS

SPRAY, AKKA & SCALA

Jan Macháček @honzam399

EASY!

```
def api(request: HttpRequest): HttpResponse =
  ???
}
```

FOR EVERY COMPLEX PROBLEM THERE IS AN ANSWER THAT IS CLEAR, SIMPLE, AND WRONG.

SPRAY'S APPROACH

A *service* is an `Actor` implementation that handles the incoming `HttpRequests`, and replies with appropriate `HttpResponses`.

```
def receive = {
  case request: HttpRequest =>
    val response = HttpResponse(...)
    sender ! response
}
```

THE REAL DEAL

```
class HelloWorldService extends Actor {

  def receive: Receive = {
    case request: HttpRequest =>
      val response = HttpResponse(...)
      sender ! response
  }

}
```

TESTING THE SERVICES

Because the service is a plain `Actor`, one can test it using TestKit.

```
class HelloWorldServiceSpec
  extends TestKit(ActorSystem())
  with SpecificationLike with ImplicitSender
  val service = TestActorRef[HelloWorldService

  "Any request" should {
    "Reply with Hello, world" in {
      service ! HttpRequest()
      expectMsgType[HttpResponse].entity
        mustEqual HttpEntity("Hello, world")
    }
  }
}
```

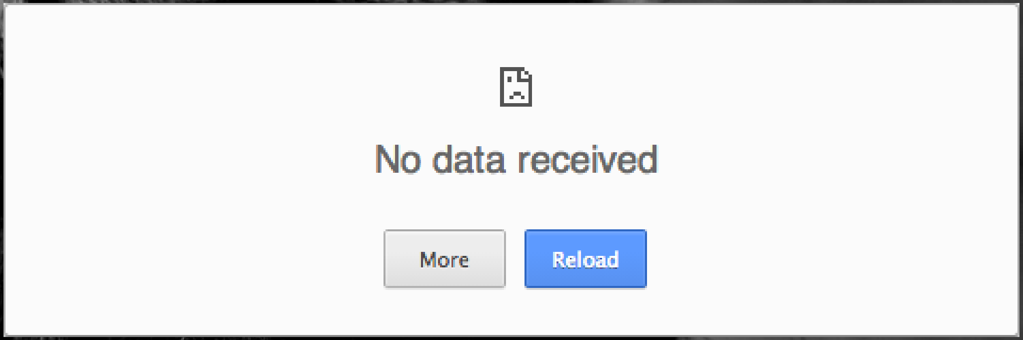
HOSTING THE SERVICES

Use `spray-can` HTTP server. We *bind* the services to it.

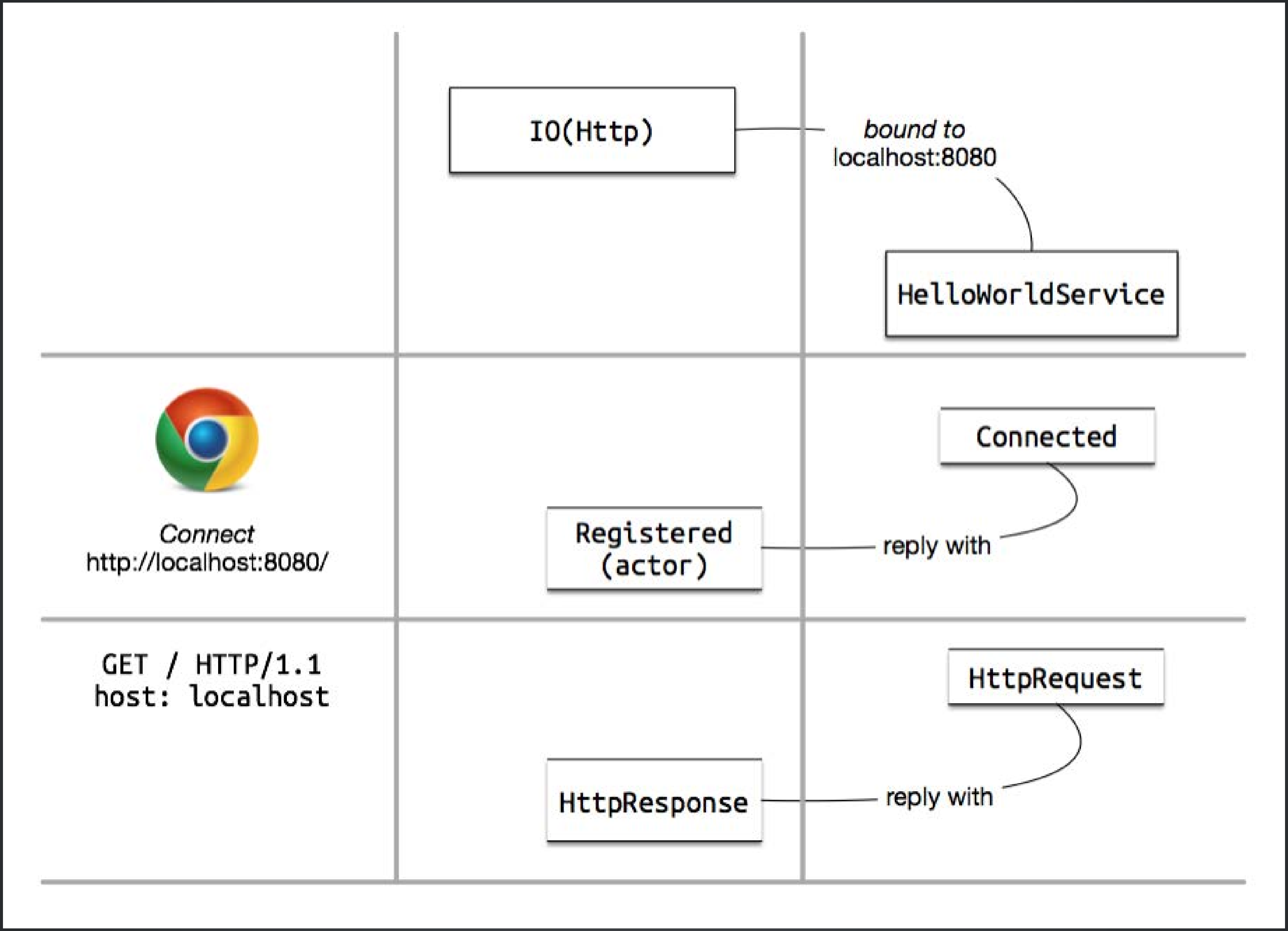
```
object HelloWorld extends App {
  val system = ActorSystem()
  val service =
    system.actorOf(Props[HelloWorldService])

  IO(Http)(system) ! Http.Bind(
    service, "0.0.0.0", port = 8080)

  Console.readLine()
  system.shutdown()
}

class HelloWorldService extends Actor { ... }
```

LET'S SEE NOW...

```
[INFO] (...) Bound to /0.0.0.0:8080
[WARN] (...) Configured registration timeout c
             second expired, stopping
```



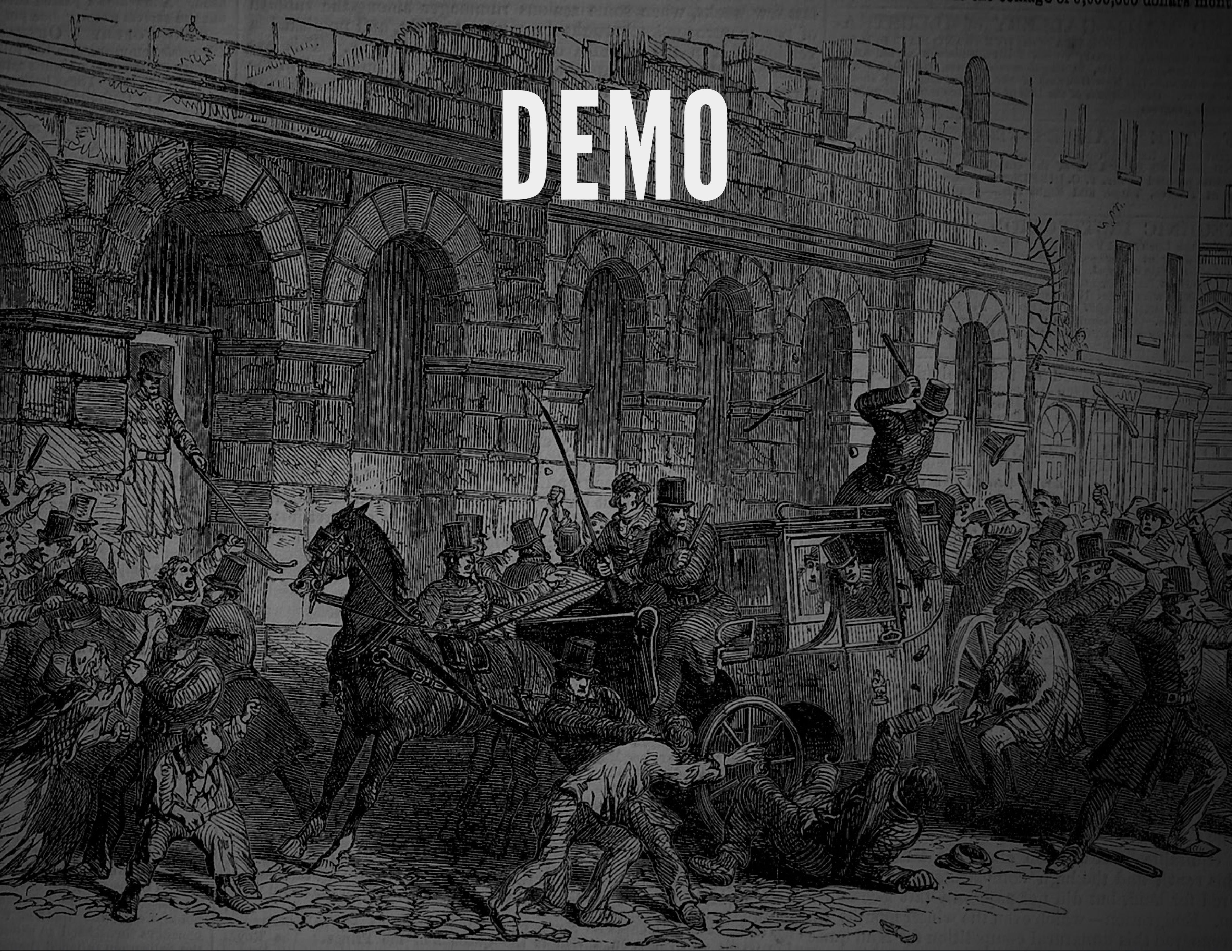
No data received


IO(Http)
bound to localhost:8080
HelloWorldService
Connected
Connect http://localhost:8080/
Registered (actor)
reply with
GET / HTTP/1.1
host: localhost
HttpRequest
HttpResponse
reply with

THE REAL DEAL

```
class HelloWorldService extends Actor {

  def receive: Receive = {
    case request: HttpRequest =>
      val response = HttpResponse(...)
      sender ! response
    case _: Http.Connected =>
      sender ! Http.Register(self)
  }

}
```

DEMO



CONVENIENT DSL

- It is tedious to build a complex API by handling the `HttpRequests`.
- Construct Spray `Routes` using convenient DSL, and then turn these routes to the `Receive` partial function.
- Use similar DSL to unit-test the routes

ROUTED HELLO, WORLD

First, the Route itself:

```
trait DemoRoute extends Directives {

  val demoRoute: Route =
    get {
      complete {
        "Hello, world"
      }
    }
}
```

ROUTED HELLO, WORLD

Next up, expressing the `Receive` PF using the route:

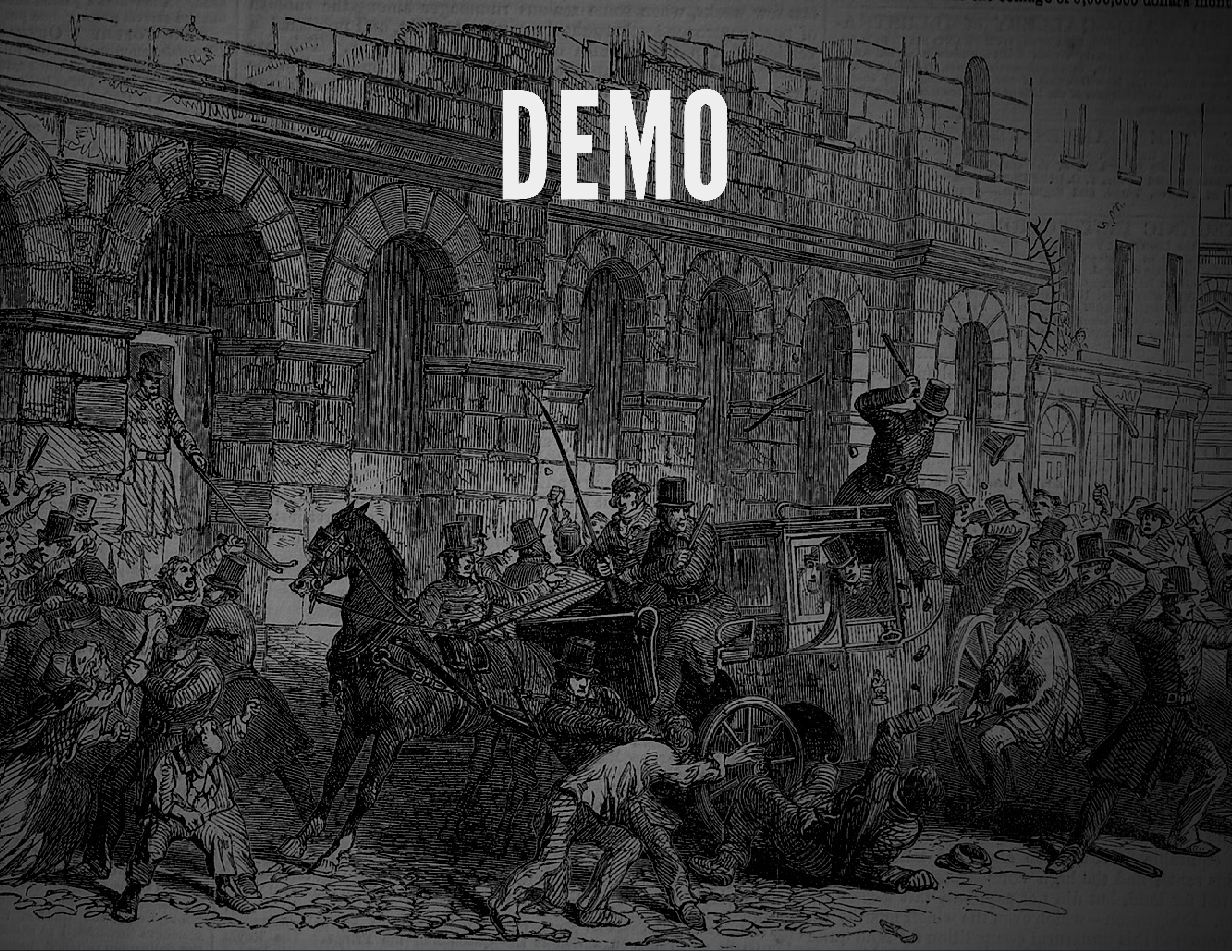
```
class MainService(route: Route)
  extends HttpServiceActor {

  def receive: Receive = runRoute(route)

}
```

TESTING OUR ROUTE

Spray's DSL extends to testing, too!

```
class DemoRouteSpec extends Specification
  with Specs2RouteTest with DemoRoute {

  "Any request" should {
    "Reply with Hello, World" in {
      Get() ~> demoRoute ~> check {
        responseAs[String] mustEqual "Hello, W
      }
    }
  }

}
```

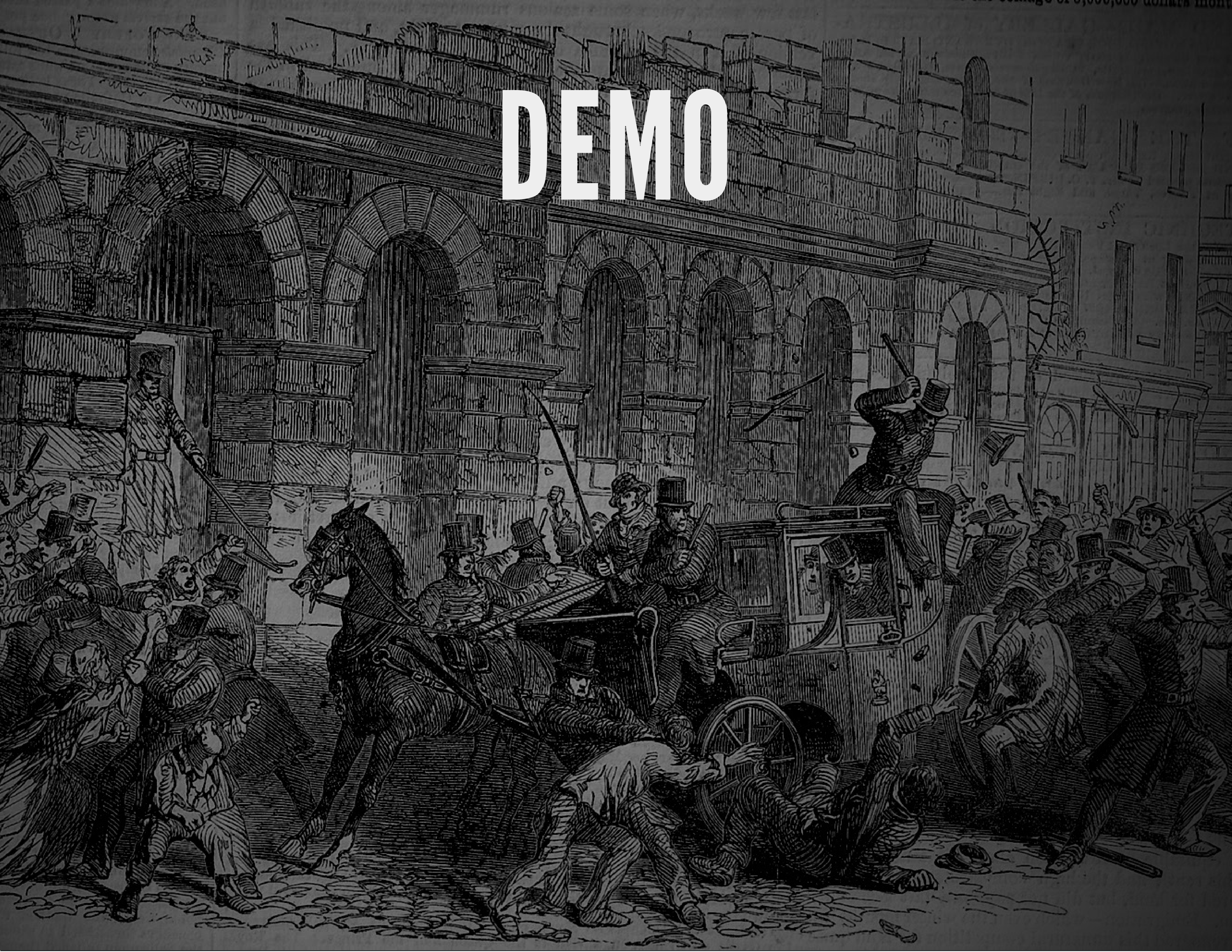
DEMO



MORE DSL EXAMPLES

- We can match—amongst others—on:
  - HTTP methods: `get`, `post`, `put`, ...,
  - Paths; including path-variables and query parameters: `path("customer" / IntNumber)`, `parameter('id.as[Int])`
  - HTTP headers: `headerValueByName("User-Agent")`,
  - Cookies: `cookie("spray")`
- We combine parts of the DSL using ~

DEMO



COMPLETING

To *complete* a route, we must provide `RequestContext => ()`. `complete` does just that, or we can do it ourselves.

```
trait TweetAnalysisRoute extends Directives {

  val tweetAnalysisRoute: Route =
    post {
      path("tweets" / Segment) ???
    }
}
```

COMPLETING

To *complete* a route, we must provide `RequestContext => ()`. `complete` does just that, or we can do it ourselves.

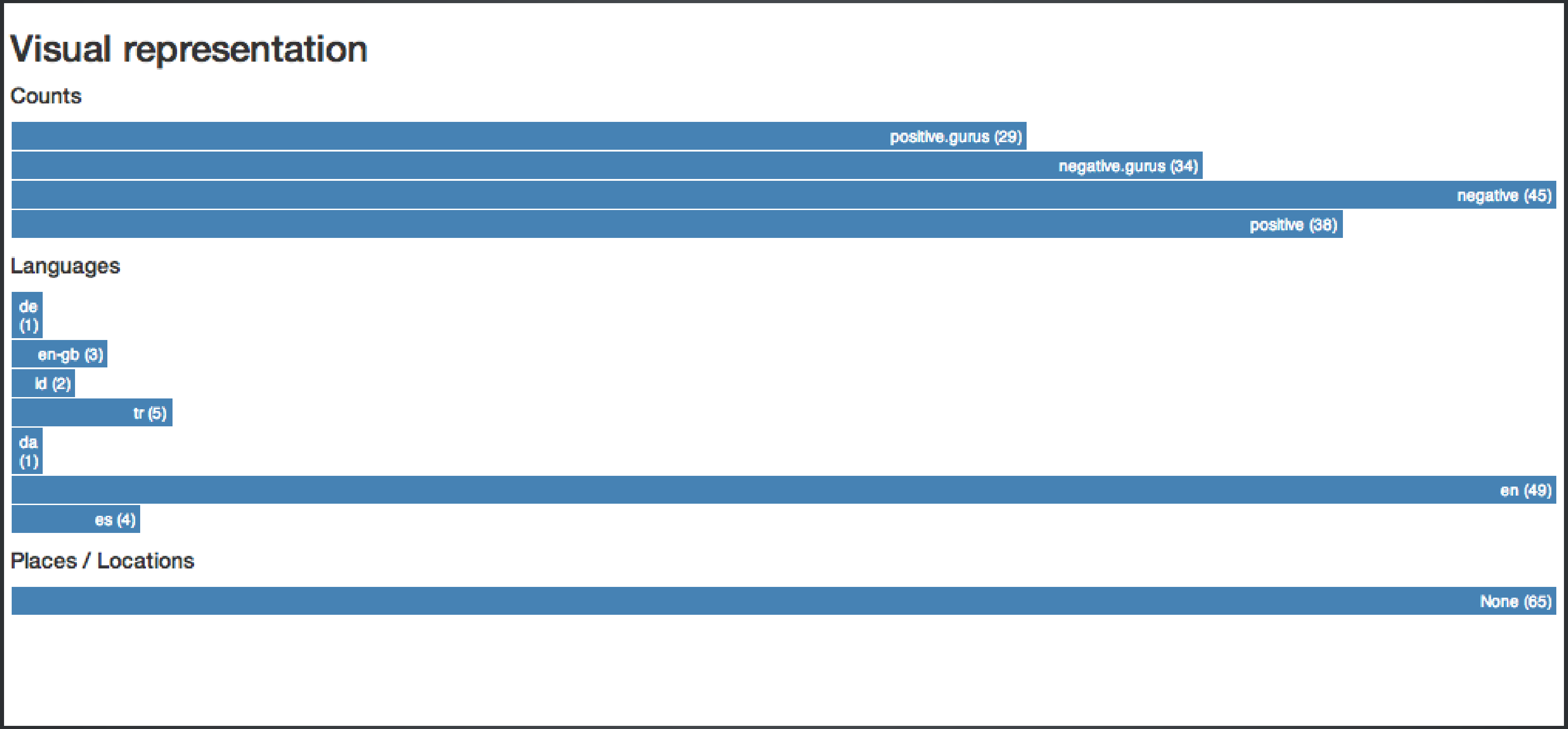
```
trait TweetAnalysisRoute extends Directives {

  val tweetAnalysisRoute: Route =
    post {
      path("tweets" / Segment)(sendTweetAnalys
    }

  def sendTweetAnalysis(query: String)
                       (ctx: RequestContext):
     ctx.receiver ! ChunkedMessageStart(...)
  }

}
```

A REAL APP

We want to stream results of a Twitter search, and show:

- Counts of *positive* and *negative* tweets,
- Counts of languages,
- Counts of locations

A REAL APP



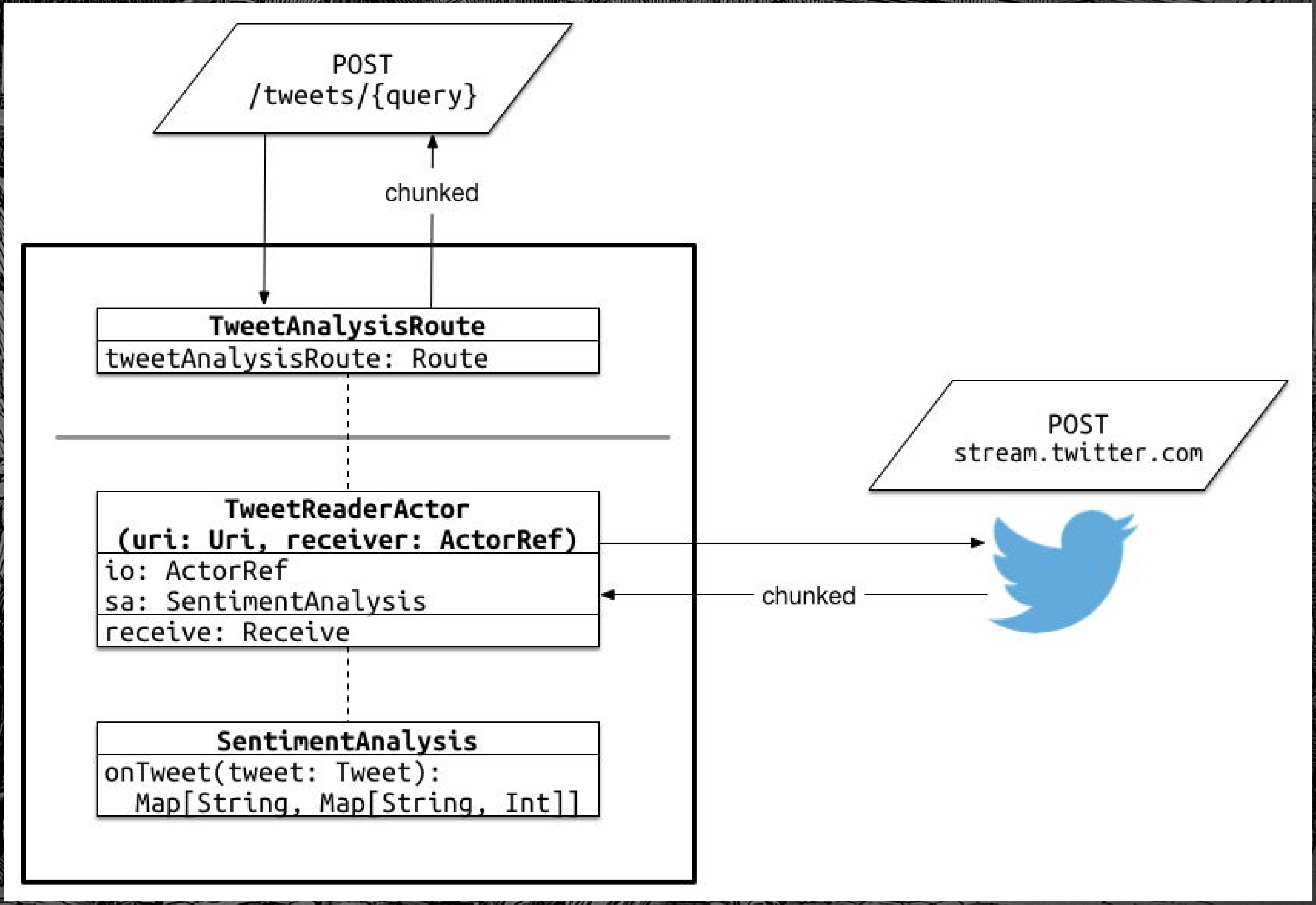
Visual representation

Counts

Languages

Places / Locations

DEMO


POST
/tweets/{query}
chunked
TweetAnalysisRoute
tweetAnalysisRoute: Route
TweetReaderActor
(uri: Uri, receiver: ActorRef)
io: ActorRef
sa: SentimentAnalysis
receive: Receive
SentimentAnalysis
onTweet(tweet: Tweet):
Map[String, Map[String, Int]]
POST
stream.twitter.com
chunked

THANK YOU!

- Source at github.com/eigengo/phillyete2014,
- Follow my on @honzam399,
- Look out for blog post at cakesolutions.net/teamblogs