HTTP Application and API

with Akka HTTP

Akka HTTP and Akka

Modern, fast, asynchronous, streaming-first HTTP server and client.

— *From https://akka.io/*

- A set of **libraries** for building applications and APIs exposed through HTTP, not a framework
- Very lightweight, runs standalone without any installation
- Very good documentation
- *Composable* with the rest of **Akka** toolkit
  - Akka **Actors**, Akka **Stream**
  - Akka Cluster / Sharding / Distributed Data / Persistence / gRPC / Management
  - Alpakka

Hello World!

```
object HelloApp {
  def main(args: Array[String]): Unit = {
    implicit val actorSystem: ActorSystem = ActorSystem("akka-http")
    implicit val executionContext: ExecutionContext = actorSystem.dispatcher
    implicit val materializer: Materializer = ActorMaterializer()

    val route = (get & path("hello") & parameter("name")) { name =>
      complete(s"Hello $name!")
    }

    val http = Http()
    http.bindAndHandle(route, "localhost", 8080)
  }
}
```

Routes and Directives

Route

- Approximately a function that takes an `HttpRequest` and produces an `HttpResponse`
- More precisely (reasonably accurate) a function that takes an `HttpRequest` and produces a `RouteResult`.
- A `RouteResult` consists of
  - either a `Complete` result containing an `HttpResponse`
  - or a `Rejected` result containing `Rejection`s

Basic Routes

```
// Completion
val hello1: Route = complete("Hello World!")

// Rejection
val rejectedHello1: Route = reject(MissingCookieRejection("username"))
```

Directive

- **Matches** an `HttpRequest`
- And optionally **extracts values** (up to 22) from it

Directive Type	Alias	Extracts
`Directive[Unit]`	`Directive0`	0 value
`Directive[Tuple1[A]]`	`Directive1[A]`	1 value of type `A`
`Directive[(A, B)]`		2 values of type `A` and `B`
`Directive[(A, B, C)]`		3 values of type `A`, `B` and `C`

Basic Directives

```
val getMethod: Directive0 = get
val helloPath: Directive0 = path("hello")
val nameParameter: Directive1[String] = parameter("name")
val idParameterAsInt: Directive1[Int] = parameter("id".as[Int])
```

Generating a `Route` from a `Directive`

```
val nameParameter: Directive1[String] = parameter("name")

val route: Route = nameParameter { name =>
  complete(s"Hello $name!")
}
```

- Combining a `Directive` and a function producing a `Route` results in a `Route`
- Arity of `Directive` directly implies arity of function

Nesting `Directives`

```
val getMethod: Directive0 = get
val helloPath: Directive0 = path("hello")
val nameParameter: Directive1[String] = parameter("name")

val route: Route = getMethod {
  helloPath {
    nameParameter { name =>
      complete(s"Hello $name!")
    }
  }
}
```

- Tries nested route only if nesting directive matches

Combining `Directives` with `&`

```
val getMethod: Directive0 = get
val helloPath: Directive0 = path("hello")
val nameParameter: Directive1[String] = parameter("name")

val directive: Directive1[String] = getMethod & helloPath & nameParameter
// arity 0 & arity 0 & arity 1 => 0 + 0 + 1 = 1
val route: Route = directive { name =>
  complete(s"Hello $name!")
}
```

- **All** combined directives have to match
- Resulting Arity is the sum of arities of combined directives

Inlining Preserves Semantics

```
val route: Route = (get & path("hello") & parameter("name")) { name =>
 complete(s"Hello $name!")
}
```

- Everything consists of **immutable values**
- **Inlining** (or **extracting**) expressions or methods does change the meaning
- Applies to `Route`, `Directive`, `PathMatcher` (more later)...

Combining `Directives` with `|`

```
val operandsPath: Directive[(Int, Int)] = path(IntNumber / IntNumber)
val operandsParameters: Directive[(Int, Int)] = parameters("a".as[Int], "b".as[Int])

val extractOperands: Directive[(Int, Int)] = operandsPath | operandsParameters
val route: Route = (pathPrefix("sum") & extractOperands) { (a, b) =>
  complete(s"$a + $b = ${a + b}")
}
```

- **Any** combined directive has to match
- First match wins
- Combined directives should have same arity
- Resulting directive will preserve arity

Combining Routes with ~

```
val hello: Route = (get & path("hello") & parameter("name")) { name =>
  complete(s"Hello $name!")
}

val sum: Route = (pathPrefix("sum") & path(IntNumber / IntNumber)) { (a, b) =>
  complete(s"$a + $b = ${a + b}")
}

val route: Route =  sum ~ route
```

- First match wins
- No match means means a rejection

Path Matchers

Matching Paths

```
// matches path similar to /sum/123/456
val sum: Route = (pathPrefix("sum") & path(IntNumber / IntNumber)) { (a, b) =>
  complete(s"$a + $b = ${a + b}")
}
```

- `pathPrefix` directive matches a **prefix** of the path,
- `path` directive matches a path until the **end of path**
- `path` and `pathPrefix` directives takes a `PathMatcher`

`pathPrefix` and `path` Directives

- **`pathPrefix` directive**
  - matches a **prefix** of the path,
  - after consuming a **leading `/`**,
  - and identifies the rest of the path
- **`path` directive**
  - matches a path until the **end of path**,
  - after consuming a **leading `/`**,
  - considering the rest of the path when `pathPrefix` was applied before

PathMatcher

- **Matches** a **prefix** of a `Path`
- And optionally **extracts values** (up to 22) from it
- And is also able to identify the **rest** of the `Path` (what remains after the matching prefix)

Path Matcher Type	Alias	Extraction
`PathMatcher[Unit]`	`PathMatcher0`	0 value
`PathMatcher[Tuple1[A]]`	`PathMatcher1[A]`	1 value of type `A`
`PathMatcher[(A, B)]`		2 values of type `A` and `B`
`PathMatcher[(A, B, C)]`		3 values of type `A`, `B` and `C`

Basic PathMatchers

```
val string: PathMatcher0 = "CLI"
val regex: PathMatcher1[String] = """[A-Z]\d{3}""".r
val segment: PathMatcher1[String] = Segment
val intNumber: PathMatcher1[Int] = IntNumber
```

Combining `PathMatchers` with `/`

```
val intNumber: PathMatcher1[Int] = IntNumber
val intNumberSlashIntNumber: PathMatcher[(Int, Int)] = intNumber / intNumber
```

- Combined path matchers match in **succession**
- Identified portions are **contiguous** and **separated by a `/`**
- Resulting Arity is the sum of arities of combined directives

Combining `PathMatchers` with `|`

```
val cli: PathMatcher0 = PathMatcher("CLI")
val cust: PathMatcher0 = "CUST"
val cliOrCust: PathMatcher0 = cli | cust
```

- **Any** combined path matcher has to match
- First match wins
- Combined path matcher should have same arity
- Resulting path matcher will preserve arity

Combining `PathMatchers` with `~`

```
val cliOrCust: PathMatcher0 = "CLI" | "CUST"
val idNumberPathMatcher: PathMatcher1[Int] = IntNumber
val customerId: PathMatcher1[Int] = cliOrCust ~ idNumberPathMatcher
```

- Combined path matchers match in **succession**
- Identified portions are **contiguous**
- Resulting Arity is the sum of arities of combined path matchers

Handling JSON

Spray JSON

a lightweight, clean and efficient JSON implementation in Scala

— *From https://github.com/spray/spray-json*

- Simple but very flexible
- Integrates seamlessly with Akka HTTP
- Heavily relies on implicits
- But no need to understand the gory details

Mapping to and from JSON

- **Model**

```
case class Cart(orderLines: Seq[OrderLine])
case class OrderLine(item: Item, quantity: Int)
case class Item(id: Int, name: String)
```

- Mapping aka. **Protocol**

```
object EcommerceProtocol extends SprayJsonSupport with DefaultJsonProtocol {
  implicit lazy val cartFormat: RootJsonFormat[Cart] = jsonFormat1(Cart.apply)
  implicit lazy val orderFormat: RootJsonFormat[OrderLine] = jsonFormat2(OrderLine.apply)
  implicit lazy val itemFormat: RootJsonFormat[Item] = jsonFormat2(Item.apply)
}
```

Mapping Sample

```
val cart = Cart(
  orderLines = Seq(
    OrderLine(item = Item(id = 1, name = "Ball"), quantity = 2),
    OrderLine(item = Item(id = 2, name = "Pen"), quantity = 1),
    OrderLine(item = Item(id = 3, name = "Fork"), quantity = 3)
  )
)
```

```
{
    "orderLines": [
        { "item": { "id": 1, "name": "Ball" }, "quantity": 2 },
        { "item": { "id": 2, "name": "Pen" }, "quantity": 1 },
        { "item": { "id": 3, "name": "Fork" }, "quantity": 3 }
    ]
}
```

Responding with JSON Object

```
val cart = Cart(
  orderLines = Seq(
    OrderLine(item = Item(id = 1, name = "Ball"), quantity = 2),
    OrderLine(item = Item(id = 2, name = "Pen"), quantity = 1),
    OrderLine(item = Item(id = 3, name = "Fork"), quantity = 3)
  )
)

import EcommerceProtocol._

val getCart = (get & path("cart")) {
  complete(cart)
}
```

Responding with JSON Array

```
val items = Seq(
  Item(id = 1, name = "Ball"),
  Item(id = 2, name = "Pen"),
  Item(id = 3, name = "Fork")
)

import EcommerceProtocol._

val getItems = (get & path("items")) {
  complete(items)
}
```

Posting JSON Object (and Array)

```
val postItem = (post & path("items")) {
  entity(as[Item]) { item =>
    complete(s"item=$item")
  }
}
```

Could also post a `Seq[Item]` using `entity(as[Seq[Item]])`

More About

- Directives
- Routes
- Testing Routes
- Twirl template engine
- sbt-revolver, a plugin for SBT enabling a super-fast development turnaround