Scala School

Лекция 14: Web сервисы

План лекции

- Play
- akka-http
- Finch/Finagle
- Http4s

Play framework

- Application layout
- Actions
- Routing
- Composing actions
- Handling errors
- Async
- Streams
- Twirl
- JSON
- Server backends

Play framework: Application layout

```
app                    → Application sources
 └ assets              → Compiled asset sources
   └ stylesheets       → Typically LESS CSS sources
   └ javascripts       → Typically CoffeeScript sources
 └ controllers         → Application controllers
 └ models              → Application business layer
 └ views               → Templates
build.sbt              → Application build script
conf                   → Configurations files and other non-compiled resources (on classpath)
 └ application.conf    → Main configuration file
 └ routes              → Routes definition
dist                   → Arbitrary files to be included in your projects distribution
public                 → Public assets
 └ stylesheets         → CSS files
 └ javascripts         → Javascript files
 └ images              → Image files
project                → sbt configuration files
 └ build.properties    → Marker for sbt project
 └ plugins.sbt         → sbt plugins including the declaration for Play itself
```

Play framework: Application layout

```
lib                    → Unmanaged libraries dependencies
logs                   → Logs folder
 └ application.log     → Default log file
target                 → Generated stuff
 └ resolution-cache    → Info about dependencies
 └ scala-2.11(2.12)
    └ api              → Generated API docs
    └ classes          → Compiled class files
    └ routes           → Sources generated from routes
    └ twirl            → Sources generated from templates
 └ universal           → Application packaging
 └ web                 → Compiled web assets
test                   → source folder for unit or functional tests
```

Play framework: Actions

```
def action = Action { implicit request =>
  anotherMethod("Some para value")
  Ok("Got request [" + request + "]")
 }

def anotherMethod(p: String)(implicit request: Request[_]) = {
 // do something that needs access to the request
 }
```

Play framework: Actions

```
import play.api.http.HttpEntity

def index = Action {
  Result(
    header = ResponseHeader(200, Map.empty),
    body = HttpEntity.Strict(ByteString("Hello world!"), Some("text/plain"))
  )
}
```

```
def index = Action {
  Ok("Hello world!")
}
```

```
val ok = Ok("Hello world!")
val notFound = NotFound
val pageNotFound = NotFound(<h1>Page not found</h1>)
val badRequest = BadRequest(views.html.form(formWithErrors))
val oops = InternalServerError("Oops")
val anyStatus = Status(488)("Strange response type")

val redirect = Redirect("/user/home")
```

Play framework: Routing

conf/routes file:

```
+ nocsrf // filter
GET   /clients/:id          controllers.Clients.show(id: Long)
GET   /files/*name          controllers.Application.download(name)
GET   /items/$id<[0-9]+>    controllers.Items.show(id: Long)
```

Play framework: Routing

conf/routes file:

```
# Extract the page parameter from the path.
GET   /:page              controllers.Application.show(page= "home")
```

Or:

```
# Extract the page parameter from the query string.
GET   /                   controllers.Application.show(page)
```

```
def show(page: String) = Action {
  loadContentFromDatabase(page).map { htmlContent =>
    Ok(htmlContent).as("text/html")
  }.getOrElse(NotFound)
}
```

Play framework: Routing

conf/routes file:

```
# Hello action
GET   /hello/:name          controllers.Application.hello(name)
```

```
// Redirect to /hello/Bob
def helloBob = Action {
  Redirect(routes.Application.hello("Bob"))
}
```

Play framework: Routing

conf/routes file:

```
# Redirects to https://www.playframework.com/ with 303 See Other
GET   /about      controllers.Default.redirect(to = "https://www.playframework.com/")

# Responds with 404 Not Found
GET   /orders     controllers.Default.notFound

# Responds with 500 Internal Server Error
GET   /clients    controllers.Default.error

# Responds with 501 Not Implemented
GET   /posts      controllers.Default.todo
```

Play framework: Composing actions

```
import play.api.mvc._

case class Logging[A](action: Action[A]) extends Action[A] {

  def apply(request: Request[A]): Future[Result] = {
    Logger.info("Calling action")
    action(request)
  }

  override def parser = action.parser
  override def executionContext = action.executionContext
}

def logging[A](action: Action[A])= Action.async(action.parser) { request =>
  Logger.info("Calling action")
  action(request)
}

def index = Logging {
  Action {
    Ok("Hello World")
  }
}
```

Play framework: Handling errors

```
import play.api.http.HttpErrorHandler
import play.api.mvc._
import play.api.mvc.Results._
import scala.concurrent._
import javax.inject.Singleton


@Singleton
class ErrorHandler extends HttpErrorHandler {

  def onClientError(request: RequestHeader, statusCode: Int, message: String) = {
    Future.successful(
      Status(statusCode)("A client error occurred: " + message)
    )
  }

  def onServerError(request: RequestHeader, exception: Throwable) = {
    Future.successful(
      InternalServerError("A server error occurred: " + exception.getMessage)
    )
  }
}
```

Play framework: Async

```
def index = Action.async {
  val futureInt = scala.concurrent.Future { intensiveComputation() }
  futureInt.map(i => Ok("Got result: " + i))
}

def echo = Action { request =>
  Ok("Got request [" + request + "]")
}

import scala.concurrent.duration._
import play.api.libs.concurrent.Futures._
def index = Action.async {
  // You will need an implicit Futures for withTimeout() -- you usually get
  // that by injecting it into your controller's constructor
  intensiveComputation().withTimeout(1.seconds).map { i =>
    Ok("Got result: " + i)
  }.recover {
    case e: scala.concurrent.TimeoutException =>
      InternalServerError("timeout")
  }
}
```

Play framework: Streams

```
def streamed = Action {

  val file = new java.io.File("/tmp/fileToServe.pdf")
  val path: java.nio.file.Path = file.toPath
  val source: Source[ByteString, _] = FileIO.fromPath(path)

  Result(
    header = ResponseHeader(200, Map.empty),
    body = HttpEntity.Streamed(source, None, Some("application/pdf"))
  )
}
```

Play framework: Streams

```
def streamed = Action {

  val file = new java.io.File("/tmp/fileToServe.pdf")
  val path: java.nio.file.Path = file.toPath
  val source: Source[ByteString, _] = FileIO.fromPath(path)

  val contentLength = Some(file.length())


  Result(
    header = ResponseHeader(200, Map.empty),
    body = HttpEntity.Streamed(source, contentLength, Some("application/pdf"))
  )
}
```

Play framework: Streams

```
def fileWithName = Action {
  Ok.sendFile(
    content = new java.io.File("/tmp/fileToServe.pdf"),
    fileName = _ => "termsOfService.pdf"
  )
}

def fileAttachment = Action {
  Ok.sendFile(
    content = new java.io.File("/tmp/fileToServe.pdf"),
    inline = false
  )
}
```

Play framework: Templates - Twirl

```
@(customer: Customer, orders: List[Order])

<h1>Welcome @customer.name!</h1>

<ul>
@for(order <- orders) {
  <li>@order.title</li>
}
</ul>
```

Play framework: Templates - Twirl

views/Application/index.scala.html:

```
@(customer: Customer, orders: List[Order])

<h1>Welcome @customer.name!</h1>

<ul>
@for(order <- orders) {
  <li>@order.title</li>
}
</ul>
```

```
val content = views.html.Application.index(c, o)
```

Play framework: Templates - Twirl

```
<ul>
@for(p <- products) {
  <li>@p.name ($@p.price)</li>
}
</ul>

@if(items.isEmpty) {
  <h1>Nothing to display</h1>
} else {
  <h1>@items.size items!</h1>
}
```

Play framework: Templates - Twirl

```
@display(product: Product) = {
  @product.name ($@product.price)
}

<ul>
@for(product <- products) {
  @display(product)
}
</ul>


@title(text: String) = @{
  text.split(' ').map(_.capitalize).mkString(" ")
}

<h1>@title("hello world")</h1>
```

Play framework: Templates - Twirl

```
@display(product: Product) = {
  @product.name ($@product.price)
}

<ul>
@for(product <- products) {
  @display(product)
}
</ul>


@title(text: String) = @{
  text.split(' ').map(_.capitalize).mkString(" ")
}

<h1>@title("hello world")</h1>
```

Play framework: Templates - Twirl

```
import play.twirl.api.StringInterpolation

val name = "Martin"
val p = html"<p>Hello $name</p>"
```

```
<ul>
  <li>@Option("value inside option")</li>
  <li>@List("first", "last")</li>
  <li>@User("Foo", "Bar")</li>
  <li>@List("hello", User("Foo", "Bar"), Option("value inside option"), List("first", "last"))</li>
</ul>
```

Play framework: Templates - Twirl

views/main.scala.html:

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <section class="content">@content</section>
  </body>
</html>
```

views/Application/index.scala.html:

```
@main(title = "Home") {

  <h1>Home page</h1>

}
```

Play framework: Body parsers

```
def save = Action { request: Request[AnyContent] =>
  val body: AnyContent = request.body
  val jsonBody: Option[JsValue] = body.asJson

  // Expecting json body
  jsonBody.map { json =>
    Ok("Got: " + (json \ "name").as[String])
  }.getOrElse {
    BadRequest("Expecting application/json request body")
  }
}
```

- **text/plain**: `String`, accessible via `asText`.
- **application/json**: `JsValue`, accessible via `asJson`.
- **application/xml**, **text/xml** or **application/XXX+xml**: `scala.xml.NodeSeq`, accessible via `asXml`.
- **application/x-www-form-urlencoded**: `Map[String, Seq[String]]`, accessible via `asFormUrlEncoded`.
- **multipart/form-data**: `MultipartFormData`, accessible via `asMultipartFormData`.
- Any other content type: `RawBuffer`, accessible via `asRaw`.

Play framework: JSON

```
case class Location(lat: Double, long: Double)

case class Place(name: String, location: Location)

object Place {

  var list: List[Place] = {
    List(
      Place(
        "Sandleford",
        Location(51.377797, -1.318965)
      ),
      Place(
        "Watership Down",
        Location(51.235685, -1.309197)
      )
    )
  }

  def save(place: Place) = {
    list = list ::: List(place)
  }
}
```

Play framework: JSON

```
import play.api.libs.json._ // JSON library
import play.api.libs.json.Reads._ // Custom validation helpers
import play.api.libs.functional.syntax._ // Combinator syntax

implicit val locationWrites: Writes[Location] = (
  (JsPath \ "lat").write[Double] and
  (JsPath \ "long").write[Double]
)(unlift(Location.unapply))

implicit val placeWrites: Writes[Place] = (
  (JsPath \ "name").write[String] and
  (JsPath \ "location").write[Location]
)(unlift(Place.unapply))

def listPlaces = Action {
  val json = Json.toJson(Place.list)
  Ok(json)
}
```

conf/routes:

```
GET   /places              controllers.Application.listPlaces
```

Play framework: JSON

```
curl --include http://localhost:9000/places

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 141

[{"name":"Sandleford","location":{"lat":51.377797,"long":-1.318965}},{"name":"Watership
Down","location":{"lat":51.235685,"long":-1.309197}}]
```

Play framework: JSON

```
implicit val locationReads: Reads[Location] = (
  (JsPath \ "lat").read[Double] and
  (JsPath \ "long").read[Double]
)(Location.apply _)

implicit val placeReads: Reads[Place] = (
  (JsPath \ "name").read[String] and
  (JsPath \ "location").read[Location]
)(Place.apply _)

def savePlace = Action(parse.json) { request =>
  val placeResult = request.body.validate[Place]
  placeResult.fold(
    errors => {
      BadRequest(Json.obj("status" ->"KO", "message" -> JsError.toJson(errors)))
    }, place => {
      Place.save(place)
      Ok(Json.obj("status" ->"OK", "message" -> ("Place '"+place.name+"' saved.") ))
    }
  )
}
```

Play framework: JSON

```
import play.api.libs.json._

implicit val residentWrites = Json.writes[Resident]

val resident = Resident(name = "Fiver", age = 4, role = None)

val residentJson: JsValue = Json.toJson(resident)
```

Play framework: Server Backends

- Akka HTTP Server
- Netty Server

Play framework: docs

- https://www.playframework.com/documentation/2.6.x/Home
- https://www.playframework.com/documentation/2.6.x/ScalaHome
- https://playframework.com/download#starters
- https://habrahabr.ru/post/319978/
- http://blog.scalac.io/2015/07/30/websockets-server-with-akka-http.html
- https://megahub.me/hub/java?w=21
- https://anadea.info/ru/blog/websockets-in-play-framework

akka-http

- Modules
- Supported Technologies
- JSON
- Routing DSL
- Examples

akka-http: Modules

Akka HTTP is structured into several modules:

- **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, this is the recommended way to write HTTP servers with Akka HTTP.

- **akka-http-core**

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

- **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**

akka-http: Supported Technologies

- HTTP
- HTTPS
- WebSocket
- HTTP/2
- Multipart
- Server-sent Events (SSE)
- JSON
- XML
- Gzip and Deflate Content-Encoding

akka-http: JSON

```scala
import akka.http.scaladsl.server.Directives
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport
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]
}
```

akka-http: JSON

```
// use it wherever json (un)marshalling is needed
class MyJsonService extends Directives with JsonSupport {
  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")
      }
    }
}
```

akka-http: Routing DSL: Routes and Directives

```
type Route = RequestContext => 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

akka-http: Routing DSL: Routes and Directives

Directives create **Routes**. To understand how directives work it is helpful to contrast them with the “primitive” way of creating routes.

Routes effectively are simply highly specialised functions that take a **RequestContext** and eventually `complete` it, which could (and often should) happen asynchronously.

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:

Scala

```
val route = complete("yeah")
```

Java

These three ways of writing this **Route** are fully equivalent, the created `route` will behave identically in all cases.

akka-http: Routing DSL: Routes and Directives

Directives create **Routes**. To understand how directives work it is helpful to contrast them with the “primitive” way of creating routes.

Routes effectively are simply highly specialised functions that take a **RequestContext** and eventually `complete` it, which could (and often should) happen asynchronously.

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:

Scala

```
val route = complete("yeah")
```

Java

These three ways of writing this **Route** are fully equivalent, the created `route` will behave identically in all cases.

akka-http: Routing DSL: Routes and Directives

```
val a = {
  println("MARK")
  complete("yeah")
}

val b = complete {
  println("MARK")
  "yeah"
}
```

akka-http: Routing DSL: Routes and Directives

```
val route: Route = { ctx =>
  if (ctx.request.method == HttpMethods.GET)
    ctx.complete("Received GET")
  else
    ctx.complete("Received something else")
}

val route =
  get {
    complete("Received GET")
  } ~
  complete("Received something else")
```

akka-http: Routing DSL: Routes and Directives

The general anatomy of a directive is as follows:

```
name(arguments) { extractions =>
  ... // inner route
}
```

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.

akka-http: Routing DSL: Composing Directives

`concat(a, b, c)` is the same as `a ~ b ~ c`.

```
val route =
  path("order" / IntNumber) { id =>
    (get | put) { ctx =>
      ctx.complete(s"Received ${ctx.request.method.name} request for order $id")
    }
  }
```

akka-http: Routing DSL: Composing Directives

```
def innerRoute(id: Int): Route =
  concat(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) }
```

akka-http: Example

```
object WebServer {
  def main(args: Array[String]) {

    implicit val system = ActorSystem("my-system")
    implicit val materializer = ActorMaterializer()
    // needed for the future flatMap/onComplete in the end
    implicit val executionContext = system.dispatcher

    val route =
      path("hello") {
        get {
          complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<h1>Say hello to akka-http</h1>"))
        }
      }
```

akka-http: Example

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

    println(s"Server online at http://localhost:8080/\nPress RETURN to stop...")
    StdIn.readLine() // let it run until user presses return
    bindingFuture
      .flatMap(_.unbind()) // trigger unbinding from the port
      .onComplete(_ => system.terminate()) // and shutdown when done
  }
}
```

akka-http: Complex example

```
object WebServer {

  // needed to run the route
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()
  // needed for the future map/flatmap in the end and future in fetchItem and saveOrder
  implicit val executionContext = system.dispatcher

  var orders: List[Item] = Nil

  // domain model
  final case class Item(name: String, id: Long)
  final case class Order(items: List[Item])
```

akka-http: Complex example

```
// formats for unmarshalling and marshalling
implicit val itemFormat = jsonFormat2(Item)
implicit val orderFormat = jsonFormat1(Order)

// (fake) async database query api
def fetchItem(itemId: Long): Future[Option[Item]] = Future {
  orders.find(o => o.id == itemId)
}
def saveOrder(order: Order): Future[Done] = {
  orders = order match {
    case Order(items) => items ::: orders
    case _            => orders
  }
  Future { Done }
}
```

akka-http: Complex example

```
def main(args: Array[String]) {

  val route: Route =
    get {
      pathPrefix("item" / LongNumber) { id =>
        // there might be no item for a given id
        val maybeItem: Future[Option[Item]] = fetchItem(id)

        onSuccess(maybeItem) {
          case Some(item) => complete(item)
          case None       => complete(StatusCodes.NotFound)
        }
      }
    } ~
```

akka-http: Complex example

```
    post {
      path("create-order") {
        entity(as[Order]) { order =>
          val saved: Future[Done] = saveOrder(order)
          onComplete(saved) { done =>
            complete("order created")
          }
        }
      }
    }

val bindingFuture = Http().bindAndHandle(route, "localhost", 8080)
println(s"Server online at http://localhost:8080/\nPress RETURN to stop...")
StdIn.readLine() // let it run until user presses return
bindingFuture
  .flatMap(_.unbind()) // trigger unbinding from the port
  .onComplete(_ ⇒ system.terminate()) // and shutdown when done  } }
```

akka-http: docs

- https://doc.akka.io/docs/akka-http/current/
- https://doc.akka.io/docs/akka-http/current/server-side/index.html
- https://habr.com/post/319978/
- https://doc.akka.io/docs/akka-http/current/routing-dsl/routes.html
- https://doc.akka.io/docs/akka-http/current/routing-dsl/case-class-extraction.html
- https://doc.akka.io/docs/akka-http/current/routing-dsl/directives/index.html
- https://doc.akka.io/docs/akka-http/current/routing-dsl/directives/alphabetically.html

Finch

- Example

Finch: Example

```
import io.finch._, io.finch.syntax._
import com.twitter.finagle.Http

val api: Endpoint[String] = get("hello") { Ok("Hello, World!") }

Http.server.serve(":8080", api.toServiceAs[Text.Plain])
```

Finch: Example

```
import com.twitter.finagle.Http
import com.twitter.util.Await

import io.finch._
import io.finch.circe._
import io.finch.syntax._
import io.circe.generic.auto._

object Main extends App {
  case class Locale(language: String, country: String)
  case class Time(locale: Locale, time: String)
  def currentTime(l: java.util.Locale): String =
    java.util.Calendar.getInstance(l).getTime.toString
  val time: Endpoint[Time] =
    post("time" :: jsonBody[Locale]) { l: Locale =>
      Ok(Time(l, currentTime(new java.util.Locale(l.language, l.country))))
    }
  Await.ready(Http.server.serve(":8081", time.toService))}
```

Finch: modules

- finch-core - the core classes/functions
- finch-generic - generic derivation for endpoints
- finch-argonaut - the JSON API support for the Argonaut library
- finch-json4s - the JSON API support for the JSON4S library
- finch-circe - the JSON API support for the Circe library
- finch-playjson - The JSON API support for the PlayJson library
- finch-sprayjson - The JSON API support for the SprayJson library
- finch-test - the test support classes/functions
- finch-sse - SSE (Server Sent Events) support in Finch

Finch: docs

- http://finagle.github.io/finch/
- https://github.com/slouc/finch-demo
- https://github.com/zdavep/finch-quickstart

Finagle: Example

```
import com.twitter.finagle.{Http, Service}
import com.twitter.finagle.http
import com.twitter.util.{Await, Future}

object Server extends App {
  val service = new Service[http.Request, http.Response] {
    def apply(req: http.Request): Future[http.Response] =
      Future.value(
        http.Response(req.version, http.Status.Ok)
      )
  }
  val server = Http.serve(":8080", service)
  Await.ready(server)
}
```

http4s

```
val service = HttpService[IO] {
  case _ =>
    IO(Response(Status.Ok))
}
// service: org.http4s.HttpService[cats.effect.IO] =
Kleisli(org.http4s.HttpService$$$Lambda$60483/1348313016@2d2cb885)

scala> val getRoot = Request[IO](Method.GET, uri("/"))
getRoot: org.http4s.Request[cats.effect.IO] = Request(method=GET, uri=/, headers=Headers())

scala> val io = service.orNotFound.run(getRoot)
io: cats.effect.IO[org.http4s.Response[cats.effect.IO]] = <function1>

scala> val response = io.unsafeRunSync
response: org.http4s.Response[cats.effect.IO] = Response(status=200, headers=Headers())
```

http4s

```
scala> val io = Ok(IO.fromFuture(IO(Future {
     |   println("I run when the future is constructed.")
     |   "Greetings from the future!"
     | })))
io: cats.effect.IO[org.http4s.Response[cats.effect.IO]] = IO$552725983

scala> io.unsafeRunSync
I run when the future is constructed.
res11: org.http4s.Response[cats.effect.IO] = Response(status=200, headers=Headers(Content-Type: text/plain;
charset=UTF-8, Content-Length: 26))
```

http4s

```
scala> val io = Ok(IO.fromFuture(IO(Future {
     |   println("I run when the future is constructed.")
     |   "Greetings from the future!"
     | })))
io: cats.effect.IO[org.http4s.Response[cats.effect.IO]] = IO$552725983

scala> io.unsafeRunSync
I run when the future is constructed.
res11: org.http4s.Response[cats.effect.IO] = Response(status=200, headers=Headers(Content-Type: text/plain;
charset=UTF-8, Content-Length: 26))
```

http4s

```
scala> HttpService[IO] {
     |   case GET -> Root / file ~ "json" => Ok(s"""{"response": "You asked for $file"}""")
     | }
res18: org.http4s.HttpService[cats.effect.IO] =
Kleisli(org.http4s.HttpService$$$Lambda$60483/1348313016@3c3a9b1e)
```

```
def getUserName(userId: Int): IO[String] = ???
// getUserName: (userId: Int)cats.effect.IO[String]

val usersService = HttpService[IO] {
  case GET -> Root / "users" / IntVar(userId) =>
    Ok(getUserName(userId))
}
// usersService: org.http4s.HttpService[cats.effect.IO] =
Kleisli(org.http4s.HttpService$$$Lambda$60483/1348313016@3030fe46)
```

http4s

```
case class User(name: String)
case class Hello(greeting: String)

implicit val decoder = jsonOf[IO, User]

val jsonService = HttpService[IO] {
  case req @ POST -> Root / "hello" =>
    for {
      // Decode a User request
      user <- req.as[User]
      // Encode a hello response
      resp <- Ok(Hello(user.name).asJson)
    } yield (resp)
}

import org.http4s.server.blaze._
val builder = BlazeBuilder[IO].bindHttp(8080).mountService(jsonService, "/").start
val blazeServer = builder.unsafeRunSync
```

http4s: docs

- https://http4s.org/v0.18/dsl/
- https://http4s.org/further-reading/

Deploy: sbt-native-packager

Build native packages for different systems

- Universal `zip`,`tar.gz`, `xz` archives
- `deb` and `rpm` packages for Debian/RHEL based systems
- `dmg` for OSX
- `msi` for Windows
- `docker` images

Provide archetypes for common use cases

- Java application with start scripts for Linux, OSX and Windows
- Java server application adds support for service managers:s
  - Systemd
  - Systemv
  - Upstart

Java8 jdkpackager wrapper

Optional JDeb integration for cross-platform Debian builds

Optional Spotify docker client integration

Deploy: sbt-native-packager

```
# universal zip
sbt universal:packageBin

# debian package
sbt debian:packageBin

# rpm package
sbt rpm:packageBin

# docker image
sbt docker:publishLocal
```

Deploy: sbt-native-packager

Installation

Add the following to your `project/plugins.sbt` file:

```
// for autoplugins
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.3.4")
```

In your `build.sbt` enable the plugin you want. For example the `JavaAppPackaging`.

```
enablePlugins(JavaAppPackaging)
```

Or if you need a server with autostart support

```
enablePlugins(JavaServerAppPackaging)
```

Спасибо