Building a REST API with Http4s

Abstracting over the effect type

ABOUT ME

GABRIEL VOLPE

- Senior Platform Engineer at Paidy
- Open Source Contributor
  - https://github.com/gvolpe
- Passionate about Functional Programming
- Ocassional writer
  - https://partialflow.wordpress.com/

WE ARE HIRING!

https://engineering.paidy.com/

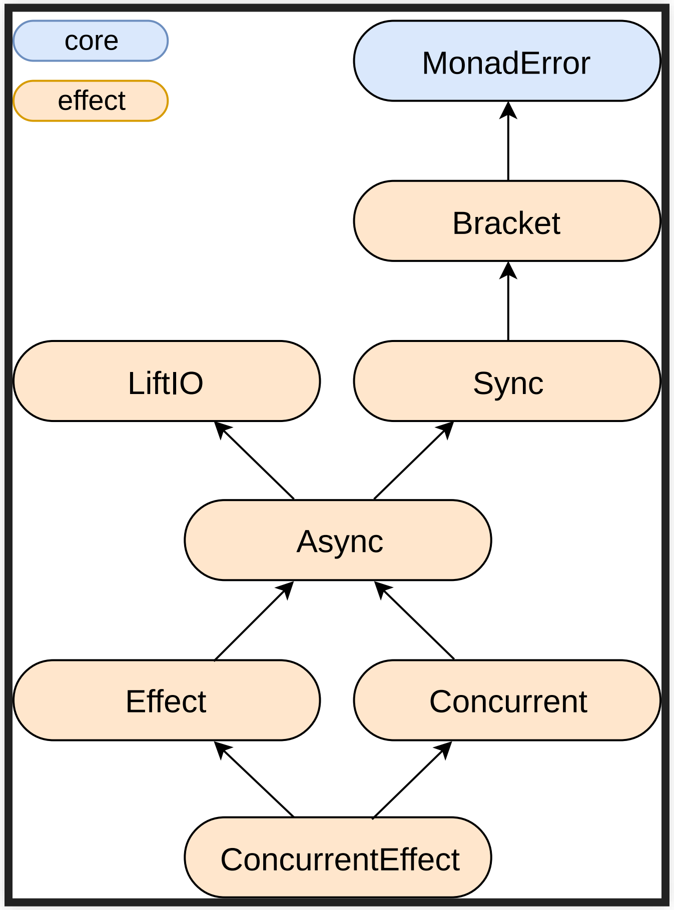
BUILDING A REST API WITH HTTP4S

AGENDA

- Basic concepts of `fs2` and `cats-effect`.
- Http4s main data types
- Json manipulation
- Error handling
- Streaming
- Live Demo

CATS EFFECT

https://typelevel.org/cats-effect/



CATS EFFECT

In addition, it provides a concrete implementation for all these typeclasses, namely the `IO` Monad.

```
import cats.effect._

val ioa: IO[Unit] = IO(println("Hello World!"))
```

This being equivalent to:

```
Sync[IO].delay(println("Hello World!"))
```

And some useful functions such as:

```
IO.fromFuture(IO {
  Future(println("Side effect!"))
})
```

FS2

Streaming library built on top of `Cats Effect`.

Its main type is:

```
class Stream[F[_], I]
```

And two other functions based on the main type:

```
type Pipe[F[_], I, O] = Stream[F, I] => Stream[F, O]
type Sink[F[_], I] = Pipe[F, I, Unit]
```

FS2

Integration with Cats Effect:

```
import cats.effect.IO
import fs2._

val stream: Stream[IO, String] = Stream.eval {  IO.pure("Hello World!") }
```

And some transformation functions:

```
val pipe: Pipe[IO, String, List[String]] = _.map(_.split(" ").toList)
```

```
val sink: Sink[IO, List[String]] = _.evalMap(x => IO(x.foreach(println)))
```

```
val program: Stream[IO, Unit] = stream through pipe to sink
```

HTTP4S MAIN DATA TYPES

`Kleisli` is at the core of Http4s.

```
type HttpService[F] = Kleisli[OptionT[F, ?], Request[F], Response[F]]
```

```
type HttpMiddleware[F[_]] = HttpService[F] => HttpService[F]
```

This is changing in dev version 0.19.

```
type Http[F[_], G[_]] = Kleisli[F, Request[G], Response[G]]
```

```
type HttpRoutes[F[_]] = Http[OptionT[F, ?], F]
```

HTTP SERVICE

We can define our endpoints as partial functions with a parametric `HttpService[F]`. In this case "/languages":

```scala
import cats.Monad
import org.http4s._
import org.http4s.dsl.Http4sDsl

class LanguagesHttpEndpoint[F[_]: Monad] extends Http4sDsl[F] {

  val service: HttpService[F] = HttpService {
    case GET -> Root / "languages" =>
      Ok(List("haskell", "idris", "scala"))
  }

}
```

Http Services are composable using the `SemigroupK` instance:

```scala
import cats.syntax.semigroupk._

val httpServices: HttpService[F] = (
  userHttpEndpoint <+> languagesHttpEndpoint
  <+> invoiceHttpEndpoint <+> salesHttpEndpoint
)
```

HTTP SERVER

```scala
import cats.effect._
import fs2.{ Stream, StreamApp }

class HttpServer[F[_]: Effect] extends StreamApp[F] {

  override def stream(args: List[String], requestShutdown: F[Unit]): Stream[F, StreamAp
    for {
      languagesHttpEndpoint <- Stream(new LanguagesHttpEndpoint[F])
      exitCode <- BlazeBuilder[F]
        .bindHttp(8080, "0.0.0.0")
        .mountService(languagesHttpEndpoint)
        .serve
    } yield exitCode

}
```

And only one place to define your concrete implementation of `cats.effect.Effect`.

```scala
object Server extends HttpServer[IO]
```

HTTP MIDDLEWARE

An Http Middleware is just a plain function:

```
type HttpMiddleware[F[_]] = HttpService[F] => HttpService[F]
```

Http4s provides some middlewares out of the box and they are composable:

```
val middleware: HttpMiddleware[F] = {
  { (service: HttpService[F]) =>
    AutoSlash(service)
  } compose { service: HttpService[F] =>
    CORS(service)
  } compose { service =>
    Timeout(2.seconds)(service)
  }
}
```

All you need to do is function application!

```
private val endpoints: HttpService[F] = ???
val httpServices: HttpService[F] = middleware(endpoints)
```

JSON CODECS

Http4s provides two interfaces for Json manipulation:

```
trait EntityDecoder[F[_], T]
```

```
trait EntityEncoder[F[_], A]
```

You can plugin your favorite Json library. However, `Circe` is the recommended one since it's integrated with `Cats` as well. You can define generic Json codecs like this for example:

```
trait JsonCodecs[F[_]] {
  implicit def jsonEncoder[A <: Product : Encoder](implicit F: Sync[F])
    : EntityEncoder[F, A] = jsonEncoderOf[F, A]
  implicit def jsonDecoder[A <: Product : Decoder](implicit F: Sync[F])
    : EntityDecoder[F, A] = jsonOf[F, A]
}
```

ERROR HANDLING: MONAD ERROR

Http4s makes use of the instance provided by `Cats Effect`.

```scala
import cats.effect.IO

val boom: IO[String] = IO.raiseError[String](new Exception("boom"))
val safe: IO[Either[Throwable, String]] = boom.attempt
```

Equivalent to:

```scala
import cats.MonadError

val M = MonadError[IO, Throwable]

val boom2: IO[String] = M.raiseError[String](new Exception("boom"))
val safe2: IO[Either[Throwable, String]] = M.attempt(boom2)
```

Handling errors:

```scala
import cats.syntax.all._

val keepGoing: IO[String] = boom.handleErrorWith {
  case e: NonFatal => IO(println(e.getMessage)) *> IO.pure("Keep going ;)")
}
```

ERROR HANDLING: HTTP RESPONSES

Given your definition of your business errors:

```scala
sealed trait ApiError extends Throwable
case class UserNotFound(username: Username) extends ApiError
case class UserAlreadyExists(username: Username) extends ApiError
```

You can make use of `handleErrorWith` to transform them into the appropiated Http Response:

```scala
class UserHttpEndpoint[F[_]](implicit M: MonadError[F, Throwable]) extends Http4sDsl[F]

  def retrieveUsers: F[List[User]] = ???

  val service: HttpService[F] = HttpService {
    case GET -> Root / "users" =>
      retrieveUsers.flatMap(users => Ok(users).handleErrorWith {
        case UserNotFound(u) => NotFound(s"User not found: ${u.value}")
        case UserAlreadyExists(u) => Conflict(s"User already exists: ${u.value}")
      }
  }

}
```

ERROR HANDLING: GENERIC HANDLER

But it's so common that makes sense to have a single error handler:

```
class HttpErrorHandler[F[_] : Monad] extends Http4sDsl[F] {

  val handle: ApiError => F[Response[F]] = {
    case UserNotFound(u) => NotFound(s"User not found: ${u.value}")
    case UserAlreadyExists(u) => Conflict(s"User already exists: ${u.value}")
  }

}
```

And make use of it in any Http Service:

```
class UserHttpEndpoint[F[_]](implicit H: HttpErrorHandler[F],
                                      M: MonadError[F, Throwable]) extends Http4sDsl[F]

  def retrieveUsers: F[List[User]] = ???

  val service: HttpService[F] = HttpService {
    case GET -> Root / "users" =>
      retrieveUsers.flatMap(users => Ok(users).handleErrorWith(H.handle)
  }

}
```

STREAMING RESPONSES

`Fs2` resides at the core of `Http4s`. Because of this, streaming responses are very simple. Consider a service returning a stream of users:

```
import fs2._

val streamOfUsers: Stream[F, User] = ??? // A call to a DB maybe?
```

And an endpoint making use of it:

```
import org.http4s._
import org.http4s.dsl._

class UserHttpEndpoint[F[_]] extends Http4sDsl[F] {

  val service: HttpService[F] = HttpService {
    case GET -> Root / "users" =>
      Ok(streamOfUsers)
  }

}
```

LIVE DEMO

https://github.com/paidy/talks

STILL MUCH MORE

Http4s comes with many other features that don't fit in a 45 mins talk:

- HttpClient[F]
  - Streaming (bracketed)
- HttpMiddleware[F]
  - CSRF
  - GZip
  - Metrics (v0.19)
- Authentication
  - Basic Auth
  - Digest Auth
  - Tsec Librabry (3rd Party)
- SSL / TLS
- Http/2 Support

カードのいらないカンタン決済

WE ARE HIRING!

https://engineering.paidy.com/