<https://blog.rockthejvm.com/http4s-tutorial/>

# Unleashing the Power of HTTP Apis: The Http4s Library

 29 minute read

This article is brought to you by [*Riccardo Cardin*](https://github.com/rcardin), a proud student of the [*Scala with Cats course*](https://rockthejvm.com/p/cats). For the last 15 years, he’s learned as much as possible about OOP, and now he is focused on his next challenge: mastering functional programming. Riccardo is a senior developer, a teacher and a passionate technical blogger.

Enter Riccardo:

Once we learned how to define Monoids, Semigroups, Applicative, Monads, and so on, it’s time to understand how to use them to build a production-ready application. Nowadays, many applications expose APIs over an HTTP channel. So, it’s worth spending some time studying libraries implementing such use case.

If we learned the basics of functional programming using the Cats ecosystem, it’s straightforward to choose the http4s library to implement HTTP endpoints. Let’s see how.

## 1. Background[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#1-background)

First things first, we need an excellent example to work with. In this case, we need a domain model easy enough to focus on creating of simple APIs.

So, imagine we’ve just finished watching the “Snyder Cut of the Justice League” movie, and we are very excited about the film. We really want to tell the world how much we enjoyed the movie, and then **we decide to build our personal “Rotten Tomatoes” application**.

As we are very experienced developers, we start coding from the backend. Hence, we begin by defining the resources we need in terms of the domain.

Indeed, we need a Movie, and a movie has a Director, many Actors, etc:

type Actor = String

case class Movie(id: String, title: String, year: Int, actors: List[String], director: String)

case class Director(firstName: String, lastName: String) {

override def toString: String = s"$firstName $lastName"

}

Without dwelling on the details, we can identify the following APIs among the others:

* Getting all movies of a director (i.e., Zack Snyder) made during a given year
* Getting the list of actors of a movie
* Adding a new director to the application

As we just finished the [Riccardo speaking here] fantastic [Cats course on Rock The JVM](https://rockthejvm.com/p/cats), we want to use a library built on the Cats ecosystem. Fortunately, the [http4s](https://http4s.org/) library is what we are looking for. So, let’s get a deep breath and start diving into the world of functional programming applied to HTTP APIs development.

## 2. Library Setup[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#2-library-setup)

The dependencies we must add in the build.sbt file are as follows:

val Http4sVersion = "1.0.0-M21"

val CirceVersion = "0.14.0-M5"

libraryDependencies ++= Seq(

"org.http4s" %% "http4s-blaze-server" % Http4sVersion,

"org.http4s" %% "http4s-circe" % Http4sVersion,

"org.http4s" %% "http4s-dsl" % Http4sVersion,

"io.circe" %% "circe-generic" % CirceVersion,

)

As version 3 of the Cats Effect library was just released, we use version 1.0.0-M21 of http4s library that integrates with it. Even though it’s not a release version, the API should be stable enough to build something. Moreover, **this version of http4s uses Scala 2.13** as the target language.

Now that we have the library dependencies, we will create a small HTTP server with the endpoints described above. Hence, we will take the following steps:

1. We will focus on route definitions and match HTTP methods, headers, path, and query parameters.
2. We will learn how to read a request body into an object and translate an object into a response body.
3. We are going to instantiate an HTTP server serving all the stuff we’ve just created.

So, let’s start the journey along with the http4s library.

## 3. Http4s Basics[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#3-http4s-basics)

**The http4s library is based on the concepts of Request and Response**. Indeed, we respond to a Request through a set of functions of type Request => Response. We call these functions routes, and **a server is nothing more than a set of routes**.

Very often, producing a Response from a Request means interacting with databases, external services, and so on, which may have some side effects. However, as diligent functional developers, **we aim to maintain the referential transparency of our functions**. Hence, the library surrounds the Response type into an effect F[\_]. So, we change the previous route definition in Request => F[Response].

Nevertheless, **not all the Request will find a route to a Response**. So, we need to take into consideration this fact, defining a route as a function of type Request => F[Option[Response]]. Using a monad transformer, we can translate this type in Request => OptionT[F, Response].

Finally, using the types Cats provides us, we can rewrite the type Request => OptionT[F, Response]using the Kleisli monad transformer. Remembering that the type Kleisli[F[\_], A, B] is just a wrapper around the function A => F[B], our route definition becomes Kleisli[OptionT[F, \*], Request, Response]. Easy, isn’t it? 😅

Fortunately, **the http4s library defines a type alias for the Kleisli monad transformer that is easier to understand for human beings: HttpRoutes[F]**.

During the tutorial, we will fill in the missing parts of our application step by step. Let’s call the application Http4sTutorial. The extensive use of the Cats library and its ecosystem requests to import many type classes and extension methods. To simplify the story, for every presented snippet of code, we will use the following imports:

import cats.\_

import cats.effect.\_

import cats.implicits.\_

import org.http4s.circe.\_

import org.http4s.\_

import io.circe.generic.auto.\_

import io.circe.syntax.\_

import org.http4s.dsl.\_

import org.http4s.dsl.impl.\_

import org.http4s.headers.\_

import org.http4s.implicits.\_

import org.http4s.server.\_

object Http4sTutorial extends IOApp {

*// ...*

}

Awesome. So, it’s time to start our journey with the implementation of the endpoint returning the list of movies associated with a particular director.

## 4. Route Definition[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#4-route-definition)

We can imagine the route that returns the list of movies of a director as something similar to the following:

GET /movies?director=Zack%20Snyder&year=2021

As we said, every route corresponds to an instance of the HttpRoutes[F] type. Again, the http4s library helps us define such routes, providing us with a dedicated DSL, the http4s-dsl.

Through the DSL, we build an HttpRoutes[F] using pattern matching as a sequence of case statements. So, let’s do it:

def movieRoutes[F[\_] : Monad]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

HttpRoutes.of[F] {

case GET -> Root / "movies" :? DirectorQueryParamMatcher(director) +& YearQueryParamMatcher(maybeYear) => ???

}

}

This structure might look extremely compact and perhaps hard to understand, but let me guide you through it real quick. **We surround the definition of all the routes in the HttpRoutes.ofconstructor**, parametrizing the routes’ definition with an effect F, as we probably have to retrieve information from some external resource.

Moreover, we import all the extension methods and implicit conversions from the Http4sDsl[F]object. The import allows the magic below to work.

Then, each case statement represents a specific route, and it matches a Request object. The DSL provides many deconstructors for a Request object, and everyone is associated with a proper unapply method.

First, the deconstructor that we can use to extract the HTTP method of a Requests and its path is called -> and decomposes them as a couple containing a Method (i.e. GET,POST, PUT, and so on), and a Path:

*// From the Http4s DSL*

object -> {

def unapply[F[\_]](req: Request[F]): Some[(Method, Path)] =

Some((req.method, Path(req.pathInfo)))

}

The definition of the route continues extracting the rest of the information of the provided URI. **If the path is absolute, we use the Root object**, which simply consumes the leading '/'character at the beginning of the path.

Each part of the remaining Path is read using a specific extractor. We use the / extractor to pattern match each piece of the path. In this case, we match the pure String "movies". However, we will see in a minute that it is possible to pattern match also directly in a variable.

### 4.1. Handling Query Parameters[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#41-handling-query-parameters)

The route we are matching contains some query parameters. **We can introduce the match of query parameters using the :? extractor**. Even though there are many ways of extracting query params, the library sponsors the use of matchers. In detail, extending the abstract class QueryParamDecoderMatcher, we can pull directly into a variable a given query parameter:

object DirectorQueryParamMatcher extends QueryParamDecoderMatcher[String]("director")

As we can see, the QueryParamDecoderMatcher requires the name of the parameter and its type. Then, the library provides the parameter’s value directly in the specified variable, which in our example is called director.

If we have to handle more than one query parameter, we can use the +& extractor, as we did in our example.

It’s possible to manage also optional query parameters. In this case, we have to change the base class of our matcher, using the OptionalQueryParamDecoderMatcher:

object YearQueryParamMatcher extends OptionalQueryParamDecoderMatcher[Year]("year")

Considering that the OptionalQueryParamDecoderMatcher class matches against optional parameters, it’s straightforward that the year variable will have the type Option[Year].

Moreover, every matcher uses inside an instance of the class QueryParamDecoder[T] to decode its query parameter. The DSL provides the decoders for basic types, such as String, numbers, etc. However, we want to interpret our "year" query parameters directly as an instance of the java.time.Year class.

To do so, starting from a QueryParamDecoder[Int], we can map the decoded result in everything we need, i.e., an instance of the java.time.Year class:

import java.time.Year

implicit val yearQueryParamDecoder: QueryParamDecoder[Year] =

QueryParamDecoder[Int].map(Year.of)

As **the matcher types access decoders using the type classes pattern**, our custom decoder must be in the scope of the matcher as an implicit value. Indeed, decoders define a companion object QueryParamDecoder that lets a matcher type summoning the proper instance of the decoder type class:

*// From the Http4s DSL*

object QueryParamDecoder {

def apply[T](implicit ev: QueryParamDecoder[T]): QueryParamDecoder[T] = ev

}

Furthermore, the companion object QueryParamDecoder contains many other practical methods to create custom decoders.

### 4.2. Matching Path Parameters[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#42-matching-path-parameters)

Another everyday use case is a route that contains some path parameters. Indeed, the API of our backend isn’t an exception, exposing a route that retrieves the list of actors of a movie:

GET /movies/aa4f0f9c-c703-4f21-8c05-6a0c8f2052f0/actors

Using the above route, we refer to a particular movie using its identifier, representing UUID. Again, the http4s library defines a set of extractors that help capture the needed information. Upgrading our movieRoutes method, for a UUID, we can use the object UUIDVar:

def movieRoutes[F[\_] : Monad]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

HttpRoutes.of[F] {

case GET -> Root / "movies" :? DirectorQueryParamMatcher(director) +& YearQueryParamMatcher(maybeYear) => ???

*// Just added*

case GET -> Root / "movies" / UUIDVar(movieId) / "actors" => ???

}

}

Hence, the variable movieId has the type java.util.UUID. Equally, the library also defines extractors for other primitive types, such as IntVar, LongVar, and the default extractor binds to a variable of type String.

However, if we need, we can develop our custom path parameter extractor, providing an object that implements the method def unapply(str: String): Option[T]. For example, if we want to extract the first and the last name of a director directly from the path, we can do the following:

object DirectorVar {

def unapply(str: String): Option[Director] = {

if (str.nonEmpty && str.matches(".\* .\*")) {

Try {

val splitStr = str.split(' ')

Director(splitStr(0), splitStr(1))

}.toOption

} else None

}

}

def directorRoutes[F[\_] : Monad]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

HttpRoutes.of[F] {

case GET -> Root / "directors" / DirectorVar(director) => ???

}

}

### 4.3. Composing Routes[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#43-composing-routes)

**The routes defined using the http4s DSL are composable**. So, it means that we can define them in different modules and then compose them in a single HttpRoutes[F] object. As developers, we know how vital module compositionality is for code that is easily maintainable and evolvable.

The trick is that the type HttpRoutes[F], being an instance of the Kleisli type, is also a [Semigroup](https://blog.rockthejvm.com/semigroups-and-monoids-in-scala/). In fact, it’s a SemigroupK, as we have a semigroup of an effect (remember the F[\_] type constructor).

The main feature of semigroups is the definition of the combine function which, given two elements of the semigroup, returns a new element also belonging to the semigroup. For the SemigroupK type class, the function is calledcombineK or <+>.

def allRoutes[F[\_] : Monad]: HttpRoutes[F] = {

import cats.syntax.semigroupk.\_

movieRoutes[F] <+> directorRoutes[F]

}

To access the <+> operator, we need the proper imports from Cats, which is at least cats.syntax.semigroupk.\_. Fortunately, we import the cats.SemigroupK type class for free using the Kleisli type.

Last but not least, an incoming request might not match with any of the available routes. Usually, such requests should end up with 404 HTTP responses. As always, the http4s library gives us an easy way to code such behavior, using the orNotFound method from the package org.http4s.implicits:

def allRoutesComplete[F[\_] : Monad]: HttpApp[F] = {

allRoutes.orNotFound

}

As we can see, the method returns an instance of the type HttpApp[F], which is a type alias for Kleisli[F, Request[F], Response[F]]. Hence, for what we said at the beginning of this article, this Kleisli is nothing more than a wrapper around the function Request[G] => F[Response[G]]. So, the difference with the HttpRoutes[F] type is that we removed the OptionT on the response.

## 5. Generate Responses[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#5-generate-responses)

As we said, **generating responses to incoming requests is an operation that could involve some side effects**. In fact, http4s handle responses with the type F[Response]. However, explicitly creating a response inside an effect F can be tedious.

So, the http4s-dsl module provides us some shortcuts to create responses associated with HTTP status codes. For example, the Ok() function creates a response with a 200 HTTP status:

val directors: mutable.Map[Actor, Director] =

mutable.Map("Zack Snyder" -> Director("Zack", "Snyder"))

def directorRoutes[F[\_] : Concurrent]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

HttpRoutes.of[F] {

case GET -> Root / "directors" / DirectorVar(director) =>

*// Just added*

directors.get(director.toString) match {

case Some(dir) => Ok(dir.asJson)

case \_ => NotFound(s"No director called $director found")

}

}

}

We will look at the meaning of the asJson method in the next section. However, it is crucial to understand the structure of the returned responses, which is:

F(

Response(

status=200,

headers=Headers(Content-Type: application/json, Content-Length: 40)

)

)

As we may imagine, inside the response, there is a place for the status, the headers, etc. However, **we don’t find the response body because the effect was not yet evaluated**.

In addition, inside the org.http4s.Status companion object, we find the functions to build a response with every other HTTP status listed in the [IANA specification](https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml).

Suppose that we want to implement some type of validation on a query parameter, as returning a Bad Request HTTP Status if the query parameter year doesn’t represent a positive number. First, we need to change the type of matcher we need to use, introducing the validation:

import scala.util.Try

implicit val yearQueryParamDecoder: QueryParamDecoder[Year] =

QueryParamDecoder[Int].emap { y =>

Try(Year.of(y))

.toEither

.leftMap { tr =>

ParseFailure(tr.getMessage, tr.getMessage)

}

}

object YearQueryParamMatcher extends OptionalValidatingQueryParamDecoderMatcher[Year]("year")

We validate the query parameter’s value using the dedicated emap method of the type QueryParamDecoder:

*// From the Http4s DSL*

def emap[U](f: T => Either[ParseFailure, U]): QueryParamDecoder[U] = ???

The function works like a common map function, but it produces an Either[ParseFailure, U] value: If the mapping succeeds, the function outputs a Right[U] value, a Left[ParseFailure] otherwise. Furthermore, the ParseFailure is a type of the http4s library indicating an error parsing an HTTP Message:

*// From the Http4s DSL*

final case class ParseFailure(sanitized: String, details: String)

The sanitized attribute may safely be displayed to a client to describe an error condition. It should not echo any part of a Request. Instead, the details attribute contains any relevant details omitted from the sanitized version of the error, and it may freely echo a Request.

Instead, the leftMap function comes as an extension method of the Cats Bifunctor type class. When the type class is instantiated for the Either type, it provides many useful methods as leftMap, which eventually applies the given function to a Left value.

Finally, the OptionalValidatingQueryParamDecoderMatcher[T] returns the result of the validation process as an instance of the type Validated[E, A]. [Validated[E, A]](https://blog.rockthejvm.com/idiomatic-error-handling-in-scala/#4-advanced-validated) is a type coming from the Cats library representing the result of a validation process: If the process ends successfully, the Validated contains instance of type A, errors of type E otherwise. Moreover, we use Validated[E, A] in opposition to Either[E, A], for example, because it accumulates errors by design. In our example, the matcher returns an instance of Validated[ParseFailure, Year].

Once validated the query parameter, we can introduce the code handling the failure with a BadRequest HTTP status:

import java.time.Year

import scala.util.Try

object DirectorQueryParamMatcher extends QueryParamDecoderMatcher[String]("director")

implicit val yearQueryParamDecoder: QueryParamDecoder[Year] =

QueryParamDecoder[Int].emap { y =>

Try(Year.of(y))

.toEither

.leftMap { tr =>

ParseFailure(tr.getMessage, tr.getMessage)

}

}

object YearQueryParamMatcher extends OptionalValidatingQueryParamDecoderMatcher[Year]("year")

def movieRoutes[F[\_] : Monad]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

HttpRoutes.of[F] {

case GET -> Root / "movies" :? DirectorQueryParamMatcher(director) +& YearQueryParamMatcher(maybeYear) =>

maybeYear match {

case Some(y) =>

y.fold(

\_ => BadRequest("The given year is not valid"),

year => ???

*// Proceeding with the business logic*

)

case None => ???

}

}

### 5.1. Headers and Cookies[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#51-headers-and-cookies)

In the previous section, we saw that http4s inserts some preconfigured headers in the response. However, it’s possible to add many other headers during the response creation:

Ok(Director("Zack", "Snyder").asJson, Header.Raw(CIString("My-Custom-Header"), "value"))

The CIString type comes from the Typelevel ecosystem (org.typelevel.ci.CIString), and represents a case-insensitive string.

In addition, the http4s library provides a lot of types in the package org.http4s.headers, representing standard HTTP headers :

import org.http4s.headers.`Content-Encoding`

Ok(`Content-Encoding`(ContentCoding.gzip))

**Cookies are nothing more than a more complex form of HTTP header**, called Set-Cookie. Again, the http4s library gives us an easy way to deal with cookies in responses. However, unlike the headers, we set the cookies after the response creation:

Ok().map(\_.addCookie(ResponseCookie("My-Cookie", "value")))

We will add more stuff to the Ok response later since we need to introduce objects’ serialization first. So, we can instantiate a new ResponseCookie, giving the constructor all the additional information needed by a cookie, such as expiration, the secure flag, httpOnly, flag, etc.

## 6. Encoding and Decoding Http Body[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#6-encoding-and-decoding-http-body)

### 6.1. Access to the Request Body[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#61-access-to-the-request-body)

By now, we should know everything we need to match routes. Now, it’s time to understand how to decode and encode structured information associated with the body of a Request or of a Response.

As we initially said, we want to let our application expose an API to add a new Director in the system. Such an API will look something similar to the following:

POST /directors

{

"firstName": "Zack",

"lastName": "Snyder"

}

First things first, to access its body, we need to access directly to the Request[F] through pattern matching:

def directorRoutes[F[\_] : Concurrent]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

HttpRoutes.of[F] {

case GET -> Root / "directors" / DirectorVar(director) =>

directors.get(director.toString) match {

case Some(dir) => Ok(dir.asJson)

case \_ => NotFound(s"No director called $director found")

}

*// Addition for inserting a new director*

case req@POST -> Root / "directors" => ???

}

}

Then, the Request[F] object has a body attribute of type EntityBody[F], which is a type alias for Stream[F, Byte]. **As the HTTP protocol defines, the body of an HTTP request is a stream of bytes**. The http4s library uses the [fs2.io](https://fs2.io/#/) library as stream implementation. Indeed, this library also uses the Typelevel stack (Cats) to implement its functional vision of streams.

### 6.2. Decoding the Request Body Using Circe[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#62-decoding-the-request-body-using-circe)

In fact, every Request[F] extends the more general Media[F] trait. This trait exposes many practical methods dealing with the body of a request, and the most interesting is the following:

*// From the Http4s DSL*

final def as[A](implicit F: MonadThrow[F], decoder: EntityDecoder[F, A]): F[A] = ???

The as function decodes a request body as a type A. using an EntityDecoder[F, A], which we must provide in the context as an implicit object.

**The EntityDecoder (and its counterpart EntityEncoder) allows us to deal with the streaming nature of the data in an HTTP body**. In addition, decoders and encoders relate directly to the Content-Type declared as an HTTP Header in the request/response. Indeed, we need a different kind of decoder and encoder for every type of content.

The http4s library ships with decoders and encoders for a limited type of contents, such as String, File, InputStream, and manages more complex contents using plugin libraries.

In our example, the request contains a new director in JSON format. **To deal with JSON body, the most frequent choice is to use the Circe plugin**.

The primary type provided by the Circe library to manipulate JSON information is the io.circe.Json type. Hence, the http4s-circe module defines the types EntityDecoder[Json] and EntityEncoder[Json], which are all we need to translate a request and a response body into an instance of Json.

However, the Json type translate directly into JSON literals, such as json"""{"firstName": "Zack", "lastName": "Snyder"}""", and we don’t really want to deal with them. Instead, we usually want a case class to represent JSON information.

First, we decode the incoming request automatically into an instance of a Json object using the EntityDecoder[Json] type. However, we need to do a step beyond to obtain an object of type Director. In detail, Circe needs an instance of the type io.circe.Decoder[Director] to decode a Json object into a Director object.

We can provide the instance of the io.circe.Decoder[Director] simply adding the import to io.circe.generic.auto.\_, which lets Circe automatically derive for us encoders and decoders. The derivation uses field names of case classes as key names in JSON objects and vice-versa. As the last step, we need to connect the type Decoder[Director] to the type EntityDecoder[Json], and this is precisely the work that the module http4s-circe does for us through the function jsonOf. Since we need the definition of the effect F in the scope, we will add the definition of the implicit value inside the method directorRoutes:

def directorRoutes[F[\_] : Concurrent]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

implicit val directorDecoder: EntityDecoder[F, Director] = jsonOf[F, Director]

*// ...*

}

Summing up, the final form of the API looks like the following:

def directorRoutes[F[\_] : Concurrent]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

implicit val directorDecoder: EntityDecoder[F, Director] = jsonOf[F, Director]

HttpRoutes.of[F] {

case GET -> Root / "directors" / DirectorVar(director) =>

directors.get(director.toString) match {

case Some(dir) => Ok(dir.asJson, Header.Raw(CIString("My-Custom-Header"), "value"))

case \_ => NotFound(s"No director called $director found")

}

case req@POST -> Root / "directors" =>

for {

director <- req.as[Director]

\_ = directors.put(director.toString, director)

res <- Ok.headers(`Content-Encoding`(ContentCoding.gzip))

.map(\_.addCookie(ResponseCookie("My-Cookie", "value")))

} yield res

}

The above code uses the for-comprehension syntax making it more readable. However, it is possible only because we initially import the Cats extension methods of cats.syntax.flatMap.\_ and cats.syntax.functor.\_. Remember that an implicit type class Monad[F] must be defined for’ F’.

Last but not least, we changed the context-bound of the effect F. In fact, the jsonOf method requires at least an instance of Concurrent to execute. So, the Monad type class is not sufficient if we need to decode a JSON request into a case class.

### 6.3. Encoding the Response Body Using Circe[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#63-encoding-the-response-body-using-circe)

The encoding process of a response body is very similar to the one described for decoding a request body. As the reference example, we go forward with the API getting all the films directed by a specific director during a year:

GET /movies?director=Zack%20Snyder&year=2021

Hence, we need to model the response of such API, which will be a list of movies with their details:

[

{

"title": "Zack Snyder's Justice League",

"year": 2021,

"actors": [

"Henry Cavill",

"Gal Godot",

"Ezra Miller",

"Ben Affleck",

"Ray Fisher",

"Jason Momoa"

],

"director": "Zack Snyder"

}

]

As for decoders, we need first to transform our class Movie into an instance of the Json type, using an instance of io.circe.Encoder[Movie]. Again, the io.circe.generic.auto.\_ import allows us to automagically define such an encoder.

Instead, we can perform the actual conversion from the Movie type to Json using the extension method asJson, defined on all the types that have an instance of the Encoder type class:

*// From the Http4s DSL*

package object syntax {

implicit final class EncoderOps[A](private val value: A) extends AnyVal {

final def asJson(implicit encoder: Encoder[A]): Json = encoder(value)

}

}

As we can see, the above method comes into the scope importing the io.circe.syntax.\_ package. As we previously said for decoders, the module http4s-circe provides the type EntityEncoder[Json], with the import org.http4s.circe.\_.

Now, we can complete our API definition, responding with the needed information.

val snjl: Movie = Movie(

"6bcbca1e-efd3-411d-9f7c-14b872444fce",

"Zack Snyder's Justice League",

2021,

List("Henry Cavill", "Gal Godot", "Ezra Miller", "Ben Affleck", "Ray Fisher", "Jason Momoa"),

"Zack Snyder"

)

val movies: Map[String, Movie] = Map(snjl.id -> snjl)

private def findMovieById(movieId: UUID) =

movies.get(movieId.toString)

private def findMoviesByDirector(director: String): List[Movie] =

movies.values.filter(\_.director == director).toList

*// Definitions of DirectorQueryParamMatcher and YearQueryParamMatcher*

def movieRoutes[F[\_] : Monad]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

HttpRoutes.of[F] {

case GET -> Root / "movies" :? DirectorQueryParamMatcher(director) +& YearQueryParamMatcher(maybeYear) =>

val movieByDirector = findMoviesByDirector(director)

maybeYear match {

case Some(y) =>

y.fold(

\_ => BadRequest("The given year is not valid"),

{ year =>

val moviesByDirAndYear =

movieByDirector.filter(\_.year == year.getValue)

Ok(moviesByDirAndYear.asJson)

}

)

case None => Ok(movieByDirector.asJson)

}

case GET -> Root / "movies" / UUIDVar(movieId) / "actors" =>

findMovieById(movieId).map(\_.actors) match {

case Some(actors) => Ok(actors.asJson)

case \_ => NotFound(s"No movie with id $movieId found")

}

}

}

## 7. Wiring All Together[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#7-wiring-all-together)

We learned how to define a route, read path and parameter variables, read and write HTTP bodies, and deal with headers and cookies. Now, it’s time to connect the pieces, defining and starting a server serving all our routes.

**The http4s library supports many types of servers**. We can look at the integration matrix directly in the [official documentation](https://http4s.org/v0.21/integrations/). However, the natively supported server is [blaze](https://github.com/http4s/blaze).

We can instantiate a new blaze server using the dedicated builder, BlazeServerBuilder, which takes as inputs the HttpRoutes (or the HttpApp), and the host and port serving the given routes:

import org.http4s.server.blaze.BlazeServerBuilder

import scala.concurrent.ExecutionContext.global

val movieApp = Http4sTutorial.allRoutesComplete[IO]

BlazeServerBuilder[IO](global)

.bindHttp(8080, "localhost")

.withHttpApp(movieApp)

// ...it's not finished yet

In blaze jargon, we say we mount the routes at the given path. The default path is "/", but if we need we can easily change the path using a Router. For example, we can partition the APIs in public (/api), and private (/api/private):

val apis = Router(

"/api" -> Http4sTutorial.movieRoutes[IO],

"/api/private" -> Http4sTutorial.directorRoutes[IO]

).orNotFound

BlazeServerBuilder[IO](global)

.bindHttp(8080, "localhost")

.withHttpApp(apis)

As we can see, the Router accepts a list of mappings from a String root to an HttpRoutes type. Moreover, we can omit the call to the bindHttp, using the port 8080 as default on the localhost address.

Moreover, the builder needs an instance of a scala.concurrent.ExecutionContext to handle incoming requests concurrently. Finally, we need to bind the builder to an effect because the execution of the service can lead to side effects itself. In our example, we bind the effect to the IO monad from the library Cats Effect.

### 7.1. Executing the Server[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#71-executing-the-server)

Once we configured the basic properties, we need to run the server. Since we chose to use the IO monad as an effect, the easier way to run an application is inside an IOApp, provided by the Cats Effect library, too.

The IOApp is a utility type that allows us to run applications that use the IO effect.

When it’s up, the server starts listening to incoming requests. **If the server stops, it must release the port and close any other resource it’s using, and this is precisely the definition of the**[**Resource**](https://typelevel.org/cats-effect/docs/std/resource)**type** from the Cats Effect library (we can think about Resources as the AutoCloseable type in Java type):

object Http4sTutorial extends IOApp {

def run(args: List[String]): IO[ExitCode] = {

val apis = Router(

"/api" -> Http4sTutorial.movieRoutes[IO],

"/api/private" -> Http4sTutorial.directorRoutes[IO]

).orNotFound

BlazeServerBuilder[IO](runtime.compute)

.bindHttp(8080, "localhost")

.withHttpApp(apis)

.resource

.use(\_ => IO.never)

.as(ExitCode.Success)

}

}

Once we have a Resource to handle, we can use its content. In this example, we say that whatever the resource is, we want to produce an effect that never terminates. In this way, the server can accept new requests over and over. Finally, the last statement maps the result of the IO into the given value, ExitCode.Success, needed by the IOApp.

### 7.2. Try It Out![Permalink](https://blog.rockthejvm.com/http4s-tutorial/#72-try-it-out)

Eventually, all the pieces should have fallen into place, and the whole code implementing our APIs is the following:

import cats.\_

import cats.effect.\_

import cats.implicits.\_

import org.http4s.circe.\_

import org.http4s.\_

import io.circe.generic.auto.\_

import io.circe.syntax.\_

import org.http4s.dsl.\_

import org.http4s.dsl.impl.\_

import org.http4s.headers.\_

import org.http4s.implicits.\_

import org.http4s.server.\_

import org.http4s.server.blaze.BlazeServerBuilder

import org.typelevel.ci.CIString

import java.time.Year

import java.util.UUID

import scala.collection.mutable

import scala.util.Try

object Http4sTutorial extends IOApp {

type Actor = String

case class Movie(id: String, title: String, year: Int, actors: List[String], director: String)

case class Director(firstName: String, lastName: String) {

override def toString: String = s"$firstName $lastName"

}

val snjl: Movie = Movie(

"6bcbca1e-efd3-411d-9f7c-14b872444fce",

"Zack Snyder's Justice League",

2021,

List("Henry Cavill", "Gal Godot", "Ezra Miller", "Ben Affleck", "Ray Fisher", "Jason Momoa"),

"Zack Snyder"

)

val movies: Map[String, Movie] = Map(snjl.id -> snjl)

object DirectorQueryParamMatcher extends QueryParamDecoderMatcher[String]("director")

implicit val yearQueryParamDecoder: QueryParamDecoder[Year] =

QueryParamDecoder[Int].emap { y =>

Try(Year.of(y))

.toEither

.leftMap { tr =>

ParseFailure(tr.getMessage, tr.getMessage)

}

}

object YearQueryParamMatcher extends OptionalValidatingQueryParamDecoderMatcher[Year]("year")

def movieRoutes[F[\_] : Monad]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

HttpRoutes.of[F] {

case GET -> Root / "movies" :? DirectorQueryParamMatcher(director) +& YearQueryParamMatcher(maybeYear) =>

val movieByDirector = findMoviesByDirector(director)

maybeYear match {

case Some(y) =>

y.fold(

\_ => BadRequest("The given year is not valid"),

{ year =>

val moviesByDirAndYear =

movieByDirector.filter(\_.year == year.getValue)

Ok(moviesByDirAndYear.asJson)

}

)

case None => Ok(movieByDirector.asJson)

}

case GET -> Root / "movies" / UUIDVar(movieId) / "actors" =>

findMovieById(movieId).map(\_.actors) match {

case Some(actors) => Ok(actors.asJson)

case \_ => NotFound(s"No movie with id $movieId found")

}

}

}

private def findMovieById(movieId: UUID) =

movies.get(movieId.toString)

private def findMoviesByDirector(director: String): List[Movie] =

movies.values.filter(\_.director == director).toList

object DirectorVar {

def unapply(str: String): Option[Director] = {

if (str.nonEmpty && str.matches(".\* .\*")) {

Try {

val splitStr = str.split(' ')

Director(splitStr(0), splitStr(1))

}.toOption

} else None

}

}

val directors: mutable.Map[Actor, Director] =

mutable.Map("Zack Snyder" -> Director("Zack", "Snyder"))

def directorRoutes[F[\_] : Concurrent]: HttpRoutes[F] = {

val dsl = Http4sDsl[F]

import dsl.\_

implicit val directorDecoder: EntityDecoder[F, Director] = jsonOf[F, Director]

HttpRoutes.of[F] {

case GET -> Root / "directors" / DirectorVar(director) =>

directors.get(director.toString) match {

case Some(dir) => Ok(dir.asJson, Header.Raw(CIString("My-Custom-Header"), "value"))

case \_ => NotFound(s"No director called $director found")

}

case req@POST -> Root / "directors" =>

for {

director <- req.as[Director]

\_ = directors.put(director.toString, director)

res <- Ok.headers(`Content-Encoding`(ContentCoding.gzip))

.map(\_.addCookie(ResponseCookie("My-Cookie", "value")))

} yield res

}

}

def allRoutes[F[\_] : Concurrent]: HttpRoutes[F] = {

movieRoutes[F] <+> directorRoutes[F]

}

def allRoutesComplete[F[\_] : Concurrent]: HttpApp[F] = {

allRoutes.orNotFound

}

import scala.concurrent.ExecutionContext.global

override def run(args: List[String]): IO[ExitCode] = {

val apis = Router(

"/api" -> Http4sTutorial.movieRoutes[IO],

"/api/private" -> Http4sTutorial.directorRoutes[IO]

).orNotFound

BlazeServerBuilder[IO](global)

.bindHttp(8080, "localhost")

.withHttpApp(apis)

.resource

.use(\_ => IO.never)

.as(ExitCode.Success)

}

}

Once the server is up and running, we can try our freshly new APIs with any HTTP client, such as cURL or something similar. Hence, the call curl 'http://localhost:8080/api/movies?director=Zack%20Snyder&year=2020' | json\_pp will produce the following response, as expected:

[

{

"id": "6bcbca1e-efd3-411d-9f7c-14b872444fce",

"director": "Zack Snyder",

"title": "Zack Snyder's Justice League",

"year": 2021,

"actors": [

"Henry Cavill",

"Gal Godot",

"Ezra Miller",

"Ben Affleck",

"Ray Fisher",

"Jason Momoa"

]

}

]

## 8. Addendum: Why We Need to Abstract over the Effect[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#8-addendum-why-we-need-to-abstract-over-the-effect)

We should have kept an eye on the use of the abstract F[\_] type constructor in the routes’ definition, instead of the use of a concrete effect type, such as IO. Let’s see why.

### 8.1. A (Very) Fast Introduction to the Effect Pattern[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#81-a-very-fast-introduction-to-the-effect-pattern)

First, why do we need to use an effect in routes definition? We must remember that we are using a library that is intended to embrace the purely functional programming paradigm. Hence, our code must adhere to the [substitution model](http://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec05.html), and we know for sure that every code producing any possible side effect is not compliant with such a model. Besides, we know that reading from a database or a file system, and in general, interacting with the outside world potentially raises exceptions or produces different results every time it is executed.

So, the functional programming paradigm overcomes this problem using the Effect Pattern. Instead of executing directly the code that may produce any side effect, **we can enclose it in a particular context that describes the possible side effect but doesn’t effectively execute it, together with the type the effect will produce**:

val hw: IO[Unit] = IO(println("Hello world!"))

For example, the IO[A] effect from the Cats Effect library represents any kind of side effect and the value of type A it might produce. The above code doesn’t print anything to the console, but it defers the execution until it’s possible. Besides, it says that it will produce a Unit value once executed. The trick is in the definition of the delay method, which is called inside the smart constructor:

def delay[A](a: => A): IO[A]

So, when will the application print the “Hello world” message to the standard output? The answer is easy: the code will be executed once the method unsafeRunSync would be called:

hw.unsafeRunSync

*// Prints "Hello world!" to the standard output*

If we want the substitution model to apply to the whole part of the application, the only appropriate place to call the unsafeRunSync is in the main method, also called the end of the world. Indeed, it’s precisely what the IOApp.run method does for us.

### 8.2. Why Binding Route Definition to a Concrete Effect Is Awful[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#82-why-binding-route-definition-to-a-concrete-effect-is-awful)

So, why don’t we directly use a concrete effect, such as the IO effect, in the routes’ definitions?

First, if we abstract the effect, **we can easily control what kind of execution model we want to apply to our server**. Will the server perform operation concurrently or sequentially? In fact, there are different kinds of effects in the Cats Effect that model the above execution models, the Sync type class for synchronous, blocking programming, and the Async type class for asynchronous, concurrent programming. Notice that the Async type extends the Sync type.

**If we need to ensure at least some properties on the execution model apply to the effect, we can use the context-bound syntax**. For example, we defined our routes handling movies using an abstract effect F that has at least an associated the Monad type class:

def movieRoutes[F[\_] : Monad]: HttpRoutes[F] = ???

We can use bound to a Monad because no operation requests anything other than sequencing functions through the flatMap method. However, the routes handling the directors need a more tight context-bound:

def directorRoutes[F[\_] : Concurrent]: HttpRoutes[F] = ???

As we said, the decoding of case classes from JSON literals through the Circe library requires some concurrency, which is expressed by the Concurrent type class coming from the Cats Effect library.

Moreover, **using a type constructor instead of a concrete effect makes the whole architecture easier to test**. Binding to a concrete effect forces us to use it in the tests, making tests more challenging to write.

For example, in tests, we can bind F[\_] to the Reader monad, instead, or any other type constructor, eventually satisfying the constraints given within the context-bound.

## 9. Conclusion[Permalink](https://blog.rockthejvm.com/http4s-tutorial/#9-conclusion)

In this article, we introduced the http4s ecosystem that helps us serving API over HTTP. The library fully embraces the functional programming paradigm, using Cats and Cats Effect as building blocks.

So, we saw how easy is the definition of routes, handling path and query parameters, and how to encode and decode JSON bodies. Finally, we wired all the things up, and we showed how to define and start a server.

Even though there many other features that this introductive tutorial doesn’t address, such as [Middlewares](https://http4s.org/v1.0/middleware/), [Streaming](https://http4s.org/v1.0/streaming/), [Authentication](https://http4s.org/v1.0/auth/), we could now build many non-trivial use cases concerning the development of HTTP API.

Hence, there is only one thing left: Try it on your own and enjoy pure functional programming HTTP APIs!