tapir documentation

Release 0.0.11

Adam Warski

Feb 22, 2023

Contents

With tapir you can describe HTTP API endpoints as immutable Scala values. Each endpoint can contain a number of input parameters, error-output parameters, and normal-output parameters. An endpoint specification can be interpreted as:

- a server, given the "business logic": a function, which computes output parameters based on input parameters. Currently supported:
  - Akka HTTP `Route`s/`Directive`s.
  - Http4s `HttpRoutes[F]`
- a client, which is a function from input parameters to output parameters. Currently supported: sttp.
- documentation. Currently supported: OpenAPI.

Tapir is licensed under Apache2, the source code is available of GitHub.

CHAPTER 1

Code teaser

```
import tapir._
import tapir.json.circe._
import io.circe.generic.auto._

type Limit = Int
type AuthToken = String
case class BooksFromYear(genre: String, year: Int)
case class Book(title: String)

val booksListing: Endpoint[(BooksFromYear, Limit, AuthToken), String, List[Book],
↪Nothing] =
  endpoint
    .get
    .in(("books" / path[String]("genre") / path[Int]("year")).mapTo(BooksFromYear))
    .in(query[Limit]("limit").description("Maximum number of books to retrieve"))
    .in(header[AuthToken]("X-Auth-Token"))
    .errorOut(stringBody)
    .out(jsonBody[List[Book]])

//

import tapir.docs.openapi._
import tapir.openapi.circe.yaml._

val docs = booksListing.toOpenAPI("My Bookshop", "1.0")
println(docs.toYaml)

//

import tapir.server.akkahttp._
import akka.http.scaladsl.server.Route
import scala.concurrent.Future

def bookListingLogic(bfy: BooksFromYear,
```

(continues on next page)

(continued from previous page)

```
                     limit: Limit,
                     at: AuthToken): Future[Either[String, List[Book]]] =
  Future.successful(Right(List(Book("The Sorrows of Young Werther"))))
val booksListingRoute: Route = booksListing.toRoute(bookListingLogic _)

//

import tapir.client.sttp._
import com.softwaremill.sttp._

val booksListingRequest: Request[Either[String, List[Book]], Nothing] = booksListing
  .toSttpRequest(uri"http://localhost:8080")
  .apply(BooksFromYear("SF", 2016), 20, "xyz-abc-123")
```

Also check out the runnable example which is a slight extension of the above.

CHAPTER 2

Contents

2.1 Quickstart

To use tapir, add the following dependency to your project:

```
"com.softwaremill.tapir" %% "tapir-core" % "0.1"
```

This will import only the core classes. To generate a server or a client, you will need to add further dependencies.

Most of tapir functionalities use package objects which provide builder and extensions methods, hence it's easiest to work with tapir if you import whole packages, e.g.:

```
import tapir._
```

If you don't have it already, you'll also need partial unification enabled in the compiler (alternatively, you'll need to manually provide type arguments in some cases). In sbt, this is:

```
scalacOptions += "-Ypartial-unification"
```

Finally, type:

```
endpoint.
```

and see where auto-complete gets you!

2.1.1 Example usages

To see an example project using Tapir, check out this Todo-Backend using tapir and http4s.

Also check out the simple runnable example which is available in the repository.

2.2 Goals of the project

- programmer-friendly, human-comprehensible types, that you are not afraid to write down
- (also inferencable by IntelliJ)
- discoverable API through standard auto-complete
- separate "business logic" from endpoint definition & documentation
- as simple as possible to generate a server, client & docs
- based purely on case class-based, immutable and reusable data structures
- first-class OpenAPI support. Provide as much or as little detail as needed.
- reasonably type safe: only, and as much types to safely generate the server/client/docs

2.2.1 Similar projects

There's a number of similar projects from which tapir draws inspiration:

- endpoints
- typedapi
- rho
- typed-schema
- guardrail

2.3 Anatomy an endpoint

An endpoint is represented as a value of type `Endpoint[I, E, O, S]`, where:

- `I` is the type of the input parameters
- `E` is the type of the error-output parameters
- `O` is the type of the output parameters
- `S` is the type of streams that are used by the endpoint's inputs/outputs

Input/output parameters (`I`, `E` and `O`) can be:

- of type `Unit`, when there's no input/ouput of the given type
- a single type
- a tuple of types

Hence, an empty, initial endpoint (`tapir.endpoint`), with no inputs and no outputs, from which all other endpoints are derived has the type:

```
val endpoint: Endpoint[Unit, Unit, Unit, Nothing] = ...
```

An endpoint which accepts two parameters of types `UUID` and `Int`, upon error returns a `String`, and on normal completion returns a `User`, would have the type:

```
Endpoint[(UUID, Int), String, User, Nothing]
```

You can think of an endpoint as a function, which takes input parameters of type `I` and returns a result of type `Either[E, O]`, where inputs or outputs can contain streaming bodies of type `S`.

2.3.1 Defining an endpoint

The description of an endpoint is an immutable case class, which includes a number of methods:

- the `name`, `description`, etc. methods allow modifying the endpoint information, which will then be included in the endpoint documentation
- the `get`, `post` etc. methods specify the HTTP method which the endpoint should support
- the `in`, `errorOut` and `out` methods allow adding a new input/output parameter
- `mapIn`, `mapInTo`, ... methods allow mapping the current input/output parameters to another value or to a case class

An important note on mapping: in tapir, all mappings are bi-directional. That's because each mapping can be used to generate a server or a client, as well as in many cases can be used both for input and for output.

2.3.2 Next

Read on about describing endpoint inputs/outputs.

2.4 Defining endpoint's input/output

An input is described by an instance of the `EndpointInput` trait, and an output by an instance of the `EndpointIO` trait, as all outputs can also be used as inputs. Each input or output can yield/accept a value. For example, `query[Int]("age"): EndpointInput[Int]` describes an input, which is the `age` query parameter, and which should be mapped (using the string-to-integer codec) as an `Int`.

The `tapir` package contains a number of convenience methods to define an input or an output for an endpoint. These are:

- `path[T]`, which captures a path segment as an input parameter of type `T`
- any string, which will be implicitly converted to a constant path segment. Path segments can be combined with the `/` method, and don't map to any values (have type `EndpointInput[Unit]`)
- `paths`, which maps to the whole remaining path as a `Seq[String]`
- `query[T](name)` captures a query parameter with the given name
- `queryParams` captures all query parameters, represented as `MultiQueryParams`
- `header[T](name)` captures a header with the given name
- `headers` captures all headers, represented as `Seq[(String, String)]`
- `cookies` captures cookies from the `Cookie` header and represents them as `List[CookiePair]`
- `setCookies` captures cookies from the `Set-Cookie` header and represents them as `List[Cookie]`
- `body[T, M]`, `stringBody`, `plainBody[T]`, `jsonBody[T]`, `binaryBody[T]`, `formBody[T]`, `multipartBody[T]` captures the body
- `streamBody[S]` captures the body as a stream: only a client/server interpreter supporting streams of type `S` can be used with such an endpoint

For outputs, you can use the `header`, `setCookies` and `body` family of methods.

2.4.1 Combining inputs and outputs

Endpoint inputs/outputs can be combined in two ways. However they are combined, the values they represent always accumulate into tuples of values.

First, descriptions can be combined using the `.and` method. Such a combination results in an input/output represented as a tuple of the given types, can be stored as a value and re-used in multiple endpoints. As all other values in tapir, endpoint input/output descriptions are immutable. For example, an input specifying two query parameters, `start` (mandatory) and `limit` (optional) can be written down as:

```
val paging: EndpointInput[(UUID, Option[Int])] =
  query[UUID]("start").and(query[Option[Int]]("limit"))

// we can now use the value in multiple endpoints, e.g.:
val listUsersEndpoint: Endpoint[(UUID, Option[Int]), Unit, List[User], Nothing] =
  endpoint.in("user" / "list").in(paging).out(jsonBody[List[User]])
```

Second, inputs can be combined by calling the `in`, `out` and `errorOut` methods on `Endpoint` multiple times. Each time such a method is invoked, it extends the list of inputs/outputs. This can be useful to separate different groups of parameters, but also to define template-endpoints, which can then be further specialized. For example, we can define a base endpoint for our API, where all paths always start with `/api/v1.0`, and errors are always returned as a json:

```
val baseEndpoint: Endpoint[Unit, ErrorInfo, Unit, Nothing] =
  endpoint.in("api" / "v1.0").errorOut(jsonBody[ErrorInfo])
```

Thanks to the fact that inputs/outputs accumulate, we can use the base endpoint to define more inputs, for example:

```
val statusEndpoint: Endpoint[Unit, ErrorInfo, Status, Nothing] =
  baseEndpoint.in("status").out(jsonBody[Status])
```

The above endpoint will correspond to the `api/v1.0/status` path.

2.4.2 Mapping over input values

Inputs/outputs can also be mapped over. As noted before, all mappings are bi-directional, so that they can be used both when interpreting an endpoint as a server, and as a client.

There's a couple of ways to map over an input/output. First, there's the `map[II](f: I => II)(g: II => I)` method, which accepts functions which provide the mapping in both directions. For example:

```
case class Paging(from: UUID, limit: Option[Int])

val paging: EndpointInput[Paging] =
  query[UUID]("start").and(query[Option[Int]]("limit"))
    .map((from, limit) => Paging(from, limit))(paging => (paging.from, paging.limit))
```

Creating a mapping between a tuple and a case class is a common operation, hence there's also a `mapTo(CaseClassCompanion)` method, which automatically provides the mapping functions:

```
case class Paging(from: UUID, limit: Option[Int])

val paging: EndpointInput[Paging] =
  query[UUID]("start").and(query[Option[Int]]("limit"))
    .mapTo(Paging)
```

Mapping methods can also be called on an endpoint (which is useful if inputs/outputs are accumulated, for example). The `Endpoint.mapIn`, `Endpoint.mapInTo` etc. have the same signatures are the ones above.

2.4.3 Next

Read on about codecs.

2.5 Codecs

A codec specifies how to map from and to raw values that are sent over the network. Raw values, which are natively supported by client/server interpreters, include `String`s, byte arrays, `File`s and multiparts.

There are built-in codecs for most common types such as `String`, `Int` etc. Codecs are usually defined as implicit values and resolved implicitly when they are referenced.

For example, a `query[Int]("quantity")` specifies an input parameter which corresponds to the `quantity` query parameter and will be mapped as an `Int`. There's an implicit `Codec[Int]` value that is referenced by the `query` method (which is defined in the `tapir` package).

In a server setting, if the value cannot be parsed as an int, a decoding failure is reported, and the endpoint won't match the request, or a `400 Bad Request` response is returned (depending on configuration).

2.5.1 Optional and multiple parameters

Some inputs/outputs allow optional, or multiple parameters:

- path segments are always required
- query and header values can be optional or multiple (repeated query parameters/headers)
- bodies can be optional, but not multiple

In general, optional parameters are represented as `Option` values, and multiple parameters as `List` values. For example, `header[Option[String]]("X-Auth-Token")` describes an optional header. An input described as `query[List[String]]("color")` allows multiple occurences of the `color` query parameter, with all values gathered into a list.

Implementation note

To support optional and multiple parameters, inputs/outputs don't require implicit `Codec` values (which represent only mandatory values), but `CodecForOptional` and `CodecForMany` implicit values.

A `CodecForOptional` can be used in a context which *allows* optional values. Given a `Codec[T]`, instances of both `CodecForOptional[T]` and `CodecForOptional[Option[T]]` will be generated (that's also the way to add support for custom optional types). The first one will require a value, and report a decoding failure if a value is missing. The second will properly map to an `Option`, depending if the value is present or not.

2.5.2 Schemas

A codec also contains the schema of the mapped type. This schema information is used when generating documentation. For primitive types, the schema values are built-in, and include values such as `Schema.SString`, `Schema.SArray`, `Schema.SBinary` etc.

For complex types, it is possible to define the schema by hand and apply it to a codec (using the `codec.schema` method), however usually the schema is looked up by codecs by requiring an implicit value of type `SchemaFor[T]`. A schema-for value contains a single `schema: Schema` field.

SchemaFor[T] values are automatically derived for case classes using Magnolia. It is possible to configure the automatic derivation to use snake-case, kebab-case or a custom field naming policy, by providing an implicit tapir.generic.Configuration value:

```
implicit val customConfiguration: Configuration =
  Configuration.defaults.snakeCaseTransformation
```

2.5.3 Media types

Codecs carry an additional type parameter, which specifies the media type. Some built-in media types include text/plain, application/json and multipart/form-data. Custom media types can be added by creating an implementation of the tapir.MediaType trait.

Thanks to codec being parametrised by media types, it is possible to have a Codec[MyCaseClass, TextPlain, _] which specifies how to serialize a case class to plain text, and a different Codec[MyCaseClass, Json, _], which specifies how to serialize a case class to json. Both can be implicitly available without implicit resolution conflicts.

Different media types can be used in different contexts. When defining a path, query or header parameter, only a codec with the TextPlain media type can be used. However, for bodies, any media types is allowed. For example, the input/output described by jsonBody[T] requires a json codec.

2.5.4 Custom types

Support for custom types can be added by writing a codec from scratch, or mapping over an existing codec. However, custom types can also be supported by mapping over inputs/outputs, not codecs. When to use one and the other?

In general, codecs should be used when translating between raw values and "application-primitives". Codecs also allow the decoding process to result in an error, or to complete successfully. For example, to support a custom id type:

```
def decode(s: String): DecodeResult[MyId] = MyId.parse(s) match {
  case Success(v) => DecodeResult.Value(v)
  case Failure(f) => DecodeResult.Error(s, f)
}
def encode(id: MyId): String = id.toString

implicit val myIdCodec: Codec[MyId] = Codec.stringPlainCodecUtf8.
↪mapDecode(decode)(encode)
```

Additionally, if a type is supported by a codec, it can be used in multiple contexts, such as query parameters, headers, bodies, etc. Mapped inputs by construction have a fixed context.

On the other hand, when building composite types out of many values, or when an isomorphic representation of a type is needed, but only for a single input/output/endpoint, mapping over an input/output is the simpler solution. Note that while codecs can report errors during decoding, mapping over inputs/outputs doesn't have this possibility.

2.5.5 Validation

While codecs support reporting decoding failures, this is not meant as a validation solution, as it only works on single values, while validation often involves multiple combined values.

Decoding failures should be reported when the input is in an incorrect low-level format, when parsing a "raw value" fails. In other words, decoding failures should be reported for format failures, not business validation errors.

Any validation should be done as part of the "business logic" methods provided to the server interpreters. In case validation fails, the result can be an error, which is one of the mappings defined in an endpoint (the `E` in `Endpoint[I, E, O, S]`).

2.5.6 Next

Read on about json support.

2.6 Working with JSON

Json values are supported through codecs which encode/decode values to json strings. However, third-party libraries are needed for actual json parsing/printing. Currently, Circe is supported. To use, add the following dependency to your project:

```
"com.softwaremill.tapir" %% "tapir-json-circe" % "0.1"
```

Next, import the package (or extend the `JsonCirce` trait, see MyTapir):

```
import tapir.json.circe._
```

This will bring into scope `Codecs` which, given an in-scope circe `Encoder`/`Decoder`, will create a codec using the json media type. Circe includes a couple of approaches to generating encoders/decoders (manual, semi-auto and auto), so you may choose whatever suits you.

For example, to automatically generate a JSON codec for a case class:

```
import tapir._
import tapir.json.circe._
import io.circe.generic.auto._

case class Book(author: String, title: String, year: Int)

val bookInput: EndpointIO[Book] = jsonBody[Book]
```

To add support for other JSON libraries, see the sources for the Circe codec (which is just a couple of lines of code).

2.6.1 Next

Read on about working with forms.

2.7 Form support

2.7.1 URL-encoded forms

An URL-encoded form input/output can be specified in two ways. First, it is possible to map all form fields as a `Seq[(String, String)]`, or `Map[String, String]` (which is more convenient if fields can't have multiple values):

```
formBody[Seq[(String, String)]]: EndpointIO[Seq[(String, String)],
                                            MediaType.XWwwFormUrlencoded, _]

formBody[Map[String, String]]: EndpointIO[Map[String, String],
                                          MediaType.XWwwFormUrlencoded, _]
```

Second, form data can be mapped to a case class. The codec for the case class is generated using a macro at compile-time. The fields of the case class should have types, for which there is a plain text codec. For example:

```
case class RegistrationForm(name: String, age: Int, news: Boolean, city:␣
↪Option[String])

formBody[RegistrationForm]
```

Each form-field is named the same as the case-class-field. The names can be transformed to snake or kebab case by providing an implicit `tapir.generic.Configuraton`.

2.7.2 Multipart forms

Similarly as above, multipart form input/outputs can be specified in two ways. To map to all parts of a multipart body, use:

```
multipartBody[Seq[AnyPart]]: EndpointIO[Seq[AnyPart], MediaType.MultipartFormData, _]
```

where `type AnyPart = Part[_]`. `Part` is a case class containing the `name` of the part, disposition parameters, headers, and the body. The bodies will be mappes as byte arrays (`Array[Byte]`), unless a custom multipart codec is defined using the `Codec.multipartCodec` method.

As with URL-encoded forms, multipart bodies can be mapped directly to case classes, however without the restriction on codecs for individual fields. Given a field of type `T`, first a plain text codec is looked up, and if one isn't found, any codec for any media type (e.g. JSON) is searched for.

Each part is named the same as the case-class-field. The names can be transformed to snake or kebab case by providing an implicit `tapir.generic.Configuraton`.

Additionally, the case class to which the multipart body is mapped can contain both normal fields, and fields of type `Part[T]`. This is useful, if part metadata (e.g. the filename) is relevant.

For example:

```
case class RegistrationForm(userData: User, photo: Part[File], news: Boolean)

multipartBody[RegistrationForm]
```

2.8 Running as an akka-http server

To expose an endpoint as an akka-http server, first add the following dependency:

```
"com.softwaremill.tapir" %% "tapir-akka-http-server" % "0.1"
```

and import the package:

```
import tapir.server.akkahttp._
```

This adds two extension methods to the `Endpoint` type: `toDirective` and `toRoute`. Both require the logic of the endpoint to be given as a function of type:

```
[I as function arguments] => Future[Either[E, O]]
```

Note that the function doesn't take the tuple `I` directly as input, but instead this is converted to a function of the appropriate arity. For example:

```
import tapir._
import tapir.server.akkahttp._
import scala.concurrent.Future
import akka.http.scaladsl.server.Route

def countCharacters(s: String): Future[Either[Unit, Int]] =
  Future.successful(Right[Unit, Int](s.length))

val countCharactersEndpoint: Endpoint[String, Unit, Int, Nothing] =
  endpoint.in(stringBody).out(plainBody[Int])

val countCharactersRoute: Route = countCharactersEndpoint.toRoute(countCharacters _)
```

The created `Route`/`Directive` can then be further combined with other akka-http directives, for example nested within other routes. The Tapir-generated `Route`/`Directive` captures from the request only what is described by the endpoint.

It's completely feasible that some part of the input is read using akka-http directives, and the rest using tapir endpoint descriptions; or, that the tapir-generated route is wrapped in e.g. a metrics route. Moreover, "edge-case endpoints", which require some special logic not expressible using tapir, can be always implemented directly using akka-http. For example:

```
val myRoute: Route = metricsDirective {
  securityDirective { user =>
    tapirEndpoint.toRoute(input => /* here we can use both `user` and `input` values␣
↪*/)
  }
}
```

2.8.1 Streaming

The akka-http interpreter accepts streaming bodies of type `Source[ByteString, Any]`, which can be used both for sending response bodies and reading request bodies. Usage: `streamBody[Source[ByteString, Any]](schema, mediaType)`.

2.8.2 Configuration

The interpreter can be configured by providing an implicit `AkkaHttpServerOptions` value and status mappers, see common server configuration for details.

2.9 Running as an http4s server

To expose an endpoint as an http4s server, first add the following dependency:

```
"com.softwaremill.tapir" %% "tapir-akka-http4s" % "0.1"
```

and import the package:

```
import tapir.server.http4s._
```

This adds an extension method to the `Endpoint` type: `toRoutes`. It requires the logic of the endpoint to be given as a function of type:

```
[I as function arguments] => F[Either[E, O]]
```

where `F[_]` is the chosen effect type. Note that the function doesn't take the tuple `I` directly as input, but instead this is converted to a function of the appropriate arity. For example:

```
import tapir._
import tapir.server.http4s._
import cats.effect.IO
import org.http4s.HttpRoutes
import cats.effect.ContextShift

// will probably come from somewhere else
implicit val cs: ContextShift[IO] =
  IO.contextShift(scala.concurrent.ExecutionContext.global)

def countCharacters(s: String): IO[Either[Unit, Int]] =
  IO.pure(Right[Unit, Int](s.length))

val countCharactersEndpoint: Endpoint[String, Unit, Int, Nothing] =
  endpoint.in(stringBody).out(plainBody[Int])
val countCharactersRoutes: HttpRoutes[IO] =
  countCharactersEndpoint.toRoutes(countCharacters _)
```

The created `HttpRoutes` are the usual http4s `Kleisli`-based transformation of a `Request` to a `Response`, and can be further composed using http4s middlewares or request-transforming functions. The tapir-generated `HttpRoutes` captures from the request only what is described by the endpoint.

It's completely feasible that some part of the input is read using a http4s wrapper function, which is then composed with the tapir endpoint descriptions. Moreover, "edge-case endpoints", which require some special logic not expressible using tapir, can be always implemented directly using http4s.

2.9.1 Streaming

The http4s interpreter accepts streaming bodies of type `Stream[F, Byte]`, which can be used both for sending response bodies and reading request bodies. Usage: `streamBody[Stream[F, Byte]](schema, mediaType)`.

2.9.2 Configuration

The interpreter can be configured by providing an implicit `Http4sServerOptions` value and status mappers, see common server configuration for details.

The http4s options also includes configuration for the blocking execution context to use, and the io chunk size.

2.10 Common server configuration

2.10.1 Status codes

By default, successful responses are returned with the `200 OK` status code, and errors with `400 Bad Request`. However, this can be customised when interpreting an endpoint as a directive/route, by providing implicit values of `type StatusMapper[T] = T => StatusCode`, where `type StatusCode = Int`.

This can be especially useful for error responses, in which case having an `Endpoint[I, E, O, S]`, you'd need to provide an implicit `StatusMapper[E]`.

2.10.2 Server options

Each interpreter accepts an implicit options value, which contains configuration values for:

- how to create a file (when receiving a response that is mapped to a file, or when reading a file-mapped multipart part)
- how to handle decode failures

Handling decode failures

Quite often user input will be malformed and decoding will fail. Should the request be completed with a `400 Bad Request` response, or should the request be forwarded to another endpoint? By default, tapir follows OpenAPI conventions, that an endpoint is uniquely identified by the method and served path. That's why:

- an "endpoint doesn't match" result is returned if the request method or path doesn't match. The http library should attempt to serve this request with the next endpoint.
- otherwise, we assume that this is the correct endpoint to serve the request, but the parameters are somehow malformed. A `400 Bad Request` response is returned if a query parameter, header or body is missing / decoding fails, or if the decoding a path capture fails with an error (but not a "missing" decode result).

This can be customised by providing an implicit instance of `tapir.server.DecodeFailureHandler`, which basing on the request, failing input and failure description can decide, whether to return a "no match", an endpoint-specific error value, or a specific response.

Only the first failure is passed to the `DecodeFailureHandler`. Inputs are decoded in the following order: method, path, query, header, body.

2.11 Using as an sttp client

Add the dependency:

```
"com.softwaremill.tapir" %% "tapir-sttp-client" % "0.1"
```

To make requests using an endpoint definition using sttp, import:

```
import tapir.client.sttp._
```

This adds the `toRequest(Uri)` extension method to any `Endpoint` instance which, given the given base URI returns a function:

```
[I as function arguments] => Request[Either[E, O], Nothing]
```

After providing the input parameters, the result is a description of the request to be made, which can be further customised and sent using any sttp backend.

See the runnable example for example usage.

2.12 Generating OpenAPI documentation

To use, add the following dependencies:

```
"com.softwaremill.tapir" %% "tapir-openapi-docs" % "0.1"
"com.softwaremill.tapir" %% "tapir-openapi-circe-yaml" % "0.1"
```

Tapir contains a case class-based model of the openapi data structures in the `openapi/openapi-model` subproject (the model is independent from all other tapir modules and can be used stand-alone).

An endpoint can be converted to an instance of the model by importing the `tapir.docs.openapi._` package and calling the provided extension method:

```
import tapir.openapi.OpenAPI
import tapir.docs.openapi._

val docs: OpenAPI = booksListing.toOpenAPI("My Bookshop", "1.0")
```

Such a model can then be refined, by adding details which are not auto-generated. Working with a deeply nested case class structure such as the `OpenAPI` one can be made easier by using a lens library, e.g. Quicklens.

The openapi case classes can then be serialised, either to JSON or YAML using Circe:

```
import tapir.openapi.circe.yaml._

println(docs.toYaml)
```

2.12.1 Exposing OpenAPI documentation

Exposing the OpenAPI documentation can be very application-specific. For example, to expose the docs using the Swagger UI and akka-http:

- add `libraryDependencies += "org.webjars" % "swagger-ui" % "3.20.5"` to `build.sbt` (or newer)
- generate the yaml content to serve as a `String` using tapir:

```
import tapir.docs.openapi._
import tapir.openapi.circe.yaml._

val docsAsYaml: String = myEndpoints.toOpenAPI("My App", "1.0").toYaml
```

- add the following routes to your server:

```
val SwaggerYml = "swagger.yml"

private val redirectToIndex: Route =
```

(continues on next page)

(continued from previous page)

```
  redirect(s"/swagger/index.html?url=/swagger/$SwaggerYml", StatusCodes.
↪PermanentRedirect)

val routes: Route =
  path("swagger") {
    redirectToIndex
  } ~
    pathPrefix("swagger") {
      path("") { // this is for trailing slash
        redirectToIndex
      } ~
        path(SwaggerYml) {
          complete(yml)
        } ~
        getFromResourceDirectory("META-INF/resources/webjars/swagger-ui/3.20.0/")
    }
```

2.13 Creating your own Tapir

Tapir uses a number of packages which contain either the data classes for describing endpoints or interpreters of this data (turning endpoints into a server or a client). Importing these packages every time you want to use Tapir may be tedious, that's why each package object inherits all of its functionality from a trait.

Hence, it is possible to create your own object which combines all of the required functionalities and provides a single-import whenever you want to use tapir. For example:

```
object MyTapir extends Tapir
  with AkkaHttpServer
  with SttpClient
  with CirceJson
  with OpenAPICirceYaml
```

Then, a single `import MyTapir._` and all Tapir data types and extensions methods will be in scope!

2.14 Contributing

Tapir is an early stage project. Everything might change. All suggestions welcome :)

See the list of issues and pick one! Or report your own.

If you are having doubts on the *why* or *how* something works, don't hesitate to ask a question on gitter or via github. This probably means that the documentation, scaladocs or code is unclear and can be improved for the benefit of all.

2.14.1 Acknowledgments

Tuple-concatenating code is copied from akka-http

Generic derivation configuration is copied from circe