tapir documentation

Release 0.x

Adam Warski

Jun 17, 2020

Getting started

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 `Routes`/`Directives`.
  - Http4s `HttpRoutes[F]`
  - Finatra `http.Controller`
- 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.

Depending on how you prefer to explore the library, take a look at one of the *examples* or read on for a more detailed description of how tapir works!

CHAPTER 1

Code teaser

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

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

//

import sttp.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 sttp.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")
```

CHAPTER 2

Other sttp projects

sttp is a family of Scala HTTP-related projects, and currently includes:

- sttp client: the Scala HTTP client you always wanted!
- sttp tapir: this project
- sttp model: simple HTTP model classes (used by client & tapir)

CHAPTER 3

Sponsors

Development and maintenance of sttp tapir is sponsored by SoftwareMill, a software development and consulting company. We help clients scale their business through software. Our areas of expertise include backends, distributed systems, blockchain, machine learning and data analytics.

CHAPTER 4

Table of contents

4.1 Quickstart

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

```
"com.softwaremill.sttp.tapir" %% "tapir-core" % "0.16.1"
```

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

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

```
import sttp.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!

4.2 Examples

The `examples` sub-project contains a number of runnable tapir usage examples:

- Hello world server, using akka-http
- Hello world server, using http4s

- Separate error & success outputs, using akka-http
- Multiple endpoints, exposing OpenAPI/Swagger documentation, using akka-http
- Multiple endpoints, exposing OpenAPI/Swagger documentation, using http4s
- Multiple endpoints, with the description coupled with server logic, using akka-http
- Reporting errors in a custom format when a query/path/.. parameter cannot be decoded
- Using custom types in endpoint descriptions
- Multipart form upload, using akka-http
- Partial server logic (authentication), using akka-http
- Books example
- ZIO example, using http4s
- ZIO partial server logic example, using http4s
- Streaming body, using akka-http
- Streaming body, using http4s + fs2

4.2.1 Other examples

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

4.2.2 Blogs, articles

- Three easy endpoints
- tAPIr's Endpoint meets ZIO's IO
- Describe, then interpret: HTTP endpoints using tapir

4.2.3 Videos

- ScalaWorld 2019: Designing Programmer-Friendly APIs

4.3 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

4.3.1 Similar projects

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

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

4.4 Basics

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`.

4.4.1 Infallible endpoints

Note that the empty `endpoint` description maps no values to either error and success outputs, however errors are still represented and allowed to occur. If you would prefer to use an endpoint description, where errors can not happen, use `infallibleEndpoint: Endpoint[Unit, Nothing, Unit, Nothing]`. This might be useful when interpreting endpoints as a client.

4.4.2 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.

4.4.3 Next

Read on about describing endpoint inputs/outputs.

4.5 Inputs/outputs

An input is described by an instance of the `EndpointInput` trait, and an output by an instance of the `EndpointOutput` trait. Some inputs can be used both as inputs and outputs; then, they additionally implement the `EndpointIO` trait.

Each input or output can yield/accept a value (but doesn't have to).

For example, `query[Int]("age")`: `EndpointInput[Int]` describes an input, which is the `age` parameter from the URI's query, and which should be coded (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. For inputs, 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 fixed 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 `List[String]`
- `query[T](name)` captures a query parameter with the given name
- `queryParams` captures all query parameters, represented as `QueryParams`
- `cookie[T](name)` captures a cookie from the `Cookie` header with the given name
- `extractFromRequest` extracts a value from the request. This input is only used by server interpreters, ignored by documentation interpreters. Client interpreters ignore the provided value.

For both inputs/outputs:

- `header[T](name)` captures a header with the given name
- `headers` captures all headers, represented as `List[Header]`
- `cookies` captures cookies from the `Cookie` header and represents them as `List[Cookie]`
- `setCookie(name)` captures the value & metadata of the a `Set-Cookie` header with a matching name
- `setCookies` captures cookies from the `Set-Cookie` header and represents them as `List[SetCookie]`
- `stringBody`, `plainBody[T]`, `jsonBody[T]`, `rawBinaryBody[R]`, `binaryBody[R, 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:

- `statusCode` maps to the status code of the response
- `statusCode(code)` maps to a fixed status code of the response

4.5.1 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, inputs/outputs can be combined using the `.and` method. Such a combination results in an input/output, which maps to a tuple of the given types. This combination can be assigned to 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.

4.5.2 Mapping over input/output 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, as well as both in input and output contexts.

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))
```

Next, you can use `mapDecode[II](f: I => DecodeResult[II])(g: II => I)`, to handle cases where decoding (mapping a low-level value to a higher-value one) can fail. There's a couple of failure reasons, captured by the alternatives of the `DecodeResult` trait.

Mappings can also be done given an `Mapping[I, II]` instance. More on that in the secion on codecs.

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 functions to construct/deconstruct the case class:

```
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.

4.5.3 Path matching

By default (as with all other types of inputs), if no path input/path segments are defined, any path will match.

If any path input/path segment is defined, the path must match *exactly* - any remaining path segments will cause the endpoint not to match the request. For example, `endpoint.in("api")` will match `/api`, `/api/`, but won't match `/`, `/api/users`.

To match only the root path, use an empty string: `endpoint.in("")` will match `http://server.com/` and `http://server.com`.

To match a path prefix, first define inputs which match the path prefix, and then capture any remaining part using `paths`, e.g.: `endpoint.in("api" / "download").in(paths)`".

4.5.4 Streaming support

Both input and output bodies can be mapped to a stream, by using `streamBody[S]`. The type `S` must match the type of streams that are supported by the interpreter: refer to the documentation of server/client interpreters for the precise type.

Adding a stream body input/output influences both the type of the input/output, as well as the 4th type parameter of `Endpoint`, which specifies the requirements regarding supported stream types for interpreters.

When using a stream body, the schema (for documentation) and format (media type) of the body must be provided by hand, as they cannot be inferred from the raw stream type. For example, to specify that the output is an akka-stream, which is a (presumably large) serialised list of json objects mapping to the `Person` class:

```
endpoint.out(streamBody[Source[ByteString, Any]](schemaFor[List[Person]], CodecFormat.
↪Json()))
```

See also the runnable streaming example.

4.5.5 Next

Read on about status codes.

4.6 Status codes

To provide a (varying) status code of a server response, use the `statusCode` output, which maps to a value of type `sttp.model.StatusCode`. The companion object contains known status codes as constants. This type of output is used only when interpreting the endpoint as a server. If your endpoint returns varying status codes which you would like to have listed in documentation use `statusCode.description(code1, "code1 description").description(code2, "code2 description")` output.

Alternatively, a fixed status code can be specified using the `statusCode(code)` output.

4.6.1 Dynamic status codes

It is also possible to specify how status codes map to different outputs. All mappings should have a common supertype, which is also the type of the output. These mappings are used to determine the status code when interpreting an endpoint as a server, as well as when generating documentation and to deserialise client responses to the appropriate type, basing on the status code.

For example, below is a specification for an endpoint where the error output is a sealed trait `ErrorInfo`; such a specification can then be refined and reused for other endpoints:

```
sealed trait ErrorInfo
case class NotFound(what: String) extends ErrorInfo
case class Unauthorized(realm: String) extends ErrorInfo
case class Unknown(code: Int, msg: String) extends ErrorInfo
case object NoContent extends ErrorInfo

// here we are defining an error output, but the same can be done for regular outputs
val baseEndpoint = endpoint.errorOut(
  oneOf[ErrorInfo](
    statusMapping(StatusCode.NotFound, jsonBody[NotFound].description("not found")),
    statusMapping(StatusCode.Unauthorized, jsonBody[Unauthorized].description(
↪"unauthorized")),
    statusMapping(StatusCode.NoContent, emptyOutput.map(_ => NoContent)(_ => ())),
    statusDefaultMapping(jsonBody[Unknown].description("unknown"))
  )
)
```

Each mapping, defined using the `statusMapping` method is a case class, containing the output description as well as the status code. Moreover, default mappings can be defined using `statusDefaultMapping`:

- for servers, the default status code for error outputs is `400`, and for normal outputs `200` (unless a `statusCode` is used in the nested output)
- for clients, a default mapping is a catch-all.

Both `statusMapping` and `statusDefaultMapping` return a value of type `StatusMapping`. A list of these values can be dynamically assembled (e.g. using a default set of cases, plus endpoint-specific mappings), and provided to `oneOf`.

4.6.2 Status mapping and type erasure

Sometime at runtime status mapping resolution can not work properly because of type erasure. For example this code will fail at compile time; because of type erasure `Right[NotFound]` and `Right[BadRequest]` will become `Right[Any]`, therefore the code would not be able to find the correct mapping for a value:

```
case class ServerError(what: String)

sealed trait UserError
case class BadRequest(what: String) extends UserError
case class NotFound(what: String) extends UserError

val baseEndpoint = endpoint.errorOut(
  oneOf[Either[ServerError, UserError]](
    statusMapping(StatusCode.NotFound, jsonBody[Right[ServerError, NotFound]].
↪description("not found")),
    statusMapping(StatusCode.BadRequest, jsonBody[Right[ServerError, BadRequest]].
↪description("unauthorized")),
    statusMapping(StatusCode.InternalServerError, jsonBody[Left[ServerError,␣
↪UserError]].description("unauthorized")),
  )
)
```

The solution is therefore to handwrite a function checking that a `val` (of type `Any`) is of the correct type:

```
val baseEndpoint = endpoint.errorOut(
  oneOf[Either[ServerError, UserError]](
    statusMappingValueMatcher(StatusCode.NotFound, jsonBody[Right[ServerError,␣
↪NotFound]].description("not found")) {
      case Right(NotFound(_)) => true
    },
    statusMappingValueMatcher(StatusCode.BadRequest, jsonBody[Right[ServerError,␣
↪BadRequest]].description("unauthorized")) {
      case Right(BadRequest(_)) => true
    },
    statusMappingValueMatcher(StatusCode.InternalServerError,␣
↪jsonBody[Left[ServerError, UserError]].description("unauthorized")) {
      case Left(ServerError(_)) => true
    }
  )
)
```

Of course you could use `statusMappingValueMatcher` to do runtime filtering for other purpose than solving type erasure.

In the case of solving type erasure, writing by hand partial function to match value against composition of case class and sealed trait can be repetitive. To make that more easy, we provide an **experimental** typeclass - `MatchType` - so you can automatically derive that partial function:

```
import sttp.tapir.typelevel.MatchType

val baseEndpoint = endpoint.errorOut(
  oneOf[Either[ServerError, UserError]](
    statusMappingFromMatchType(StatusCode.NotFound, jsonBody[Right[ServerError,␣
↪NotFound]].description("not found")),
    statusMappingFromMatchType(StatusCode.BadRequest, jsonBody[Right[ServerError,␣
↪BadRequest]].description("unauthorized")),
    statusMappingFromMatchType(StatusCode.InternalServerError,␣
↪jsonBody[Left[ServerError, UserError]].description("unauthorized"))
  )
)
```

4.6.3 Server interpreters

Unless specified otherwise, successful responses are returned with the `200 OK` status code, and errors with `400 Bad Request`. For exception and decode failure handling, see error handling.

4.6.4 Next

Read on about codecs.

4.7 Codecs

4.7.1 Mappings

The `Mapping[L, H]` trait defines a bi-directional mapping between values of type `L` and values of type `H`.

Low-level values of type `L` can be **decoded** to a higher-level value of type `H`. The decoding can fail; this is represented by a result of type `DecodeResult.Failure`. Failures might occur due to format errors, wrong arity, exceptions, or validation errors. Validators can be added through the `validate` method.

High-level values of type `H` can be **encoded** as a low-level value of type `L`.

Mappings can be chained using one of the `map` functions.

4.7.2 Codecs

A `Codec[L, H, CF]` is a `Mapping[L, H]`, with additional meta-data: an optional schema and the format of the low-level value (more on that below).

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. However, they can also be provided explicitly as needed.

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[List[String], Int, TextPlain]` value that is referenced by the `query` method (which is defined in the `sttp.tapir` package).

In this example, the low-level value is a `List[String]`, as a given query parameter can be absent, have a single or many values. The high-level value is an `Int`. The codec will verify that there's a single query paramater with the given name, and parse it as an int. If any of this fails, a failure will be reported.

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).

As each codec is also a `Mapping`, codecs can be used to map endpoint inputs and outputs. In such cases, the additional meta-data (schema & format) isn't taken into account.

4.7.3 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 occurrences of the `color` query parameter, with all values gathered into a list.

4.7.4 Schemas

A codec contains an optional schema, which describes how the high-level value is encoded when sent over the network. This schema information is used when generating documentation.

Schemas consists of the schema type, which is one of the values defined in `SchemaType`, such as `SString`, `SBinary`, `SArray` or `SProduct` (for objects). Moreover, a schema contains meta-data: value optionality, description and low-level format.

For primitive types, the schema values are built-in, and defined in the `Schema` companion object.

The schema is left unchanged when mapping a codec or an input/output, as the underlying representation of the value doesn't change. However, schemas can be changed for individual inputs/outputs using the `.schema(Schema)` method.

When codecs are derived for complex types, e.g. for json mapping, schemas are looked up through implicit `Schema[T]` values. See custom types for more details.

4.7.5 Codec format

Codecs contain an additional type parameter, which specifies the codec format. Each format corresponds to a media type, which describes the low-level format of the raw value (to which the codec encodes). Some built-in formats include `text/plain`, `application/json` and `multipart/form-data`. Custom formats can be added by creating an implementation of the `sttp.tapir.CodecFormat` trait.

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

Different codec formats 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.

4.7.6 Next

Read on about custom types.

4.8 Custom types

To support a custom type, you'll need to provide an implicit `Codec` for that type.

This can be done by writing a codec from scratch, mapping over an existing codec, or automatically deriving one. Which of these approaches can be taken, depends on the context in which the codec will be used.

4.8.1 Providing an implicit codec

To create a custom codec, you can either directly implement the `Codec` trait, which requires to provide the following information:

- `encode` and `rawDecode` methods
- optional schema (for documentation)
- optional validator
- codec format (`text/plain`, `application/json` etc.)

This might be quite a lot of work, that's why it's usually easier to map over an existing codec. To do that, you'll need to provide two mappings:

- a `decode` method which decodes the lower-level type into the custom type, optionally reporting decode failures (the return type is a `DecodeResult`)
- an `encode` method which encodes the custom type into the lower-level type

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[String, MyId, TextPlain] =
  Codec.string.mapDecode(decode)(encode)

// or, using the type alias for codecs in the TextPlain format and String as the raw␣
↪value:
implicit val myIdCodec: PlainCodec[MyId] = Codec.string.mapDecode(decode)(encode)
```

Note: Note that inputs/outputs can also be mapped over. In some cases, it's enough to create an input/output corresponding to one of the existing types, and then map over them. However, if you have a type that's used multiple times, it's usually better to define a codec for that type.

Then, you can use the new codec e.g. to obtain an id from a query parameter or a path segment:

```
endpoint.in(query[MyId]("myId"))
// or
endpoint.in(path[MyId])
```

4.8.2 Automatically deriving codecs

In some cases, codecs can be automatically derived:

- for supported json libraries
- for urlencoded and multipart forms

Automatic codec derivation usually requires other implicits, such as:

- json encoders/decoders from the json library
- codecs for individual form fields

- schema of the custom type, through the `Schema[T]` implicit

4.8.3 Schema derivation

For case classes types, `Schema[_]` values are derived automatically using Magnolia, given that schemas are defined for all of the case class's fields. 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.default.withSnakeCaseMemberNames
```

Alternatively, `Schema[_]` values can be defined by hand, either for whole case classes, or only for some of its fields. For example, here we state that the schema for `MyCustomType` is a `String`:

```
implicit val schemaForMyCustomType: Schema[MyCustomType] = Schema(SchemaType.SString)
```

If you have a case class which contains some non-standard types (other than strings, number, other case classes, collections), you only need to provide the schema for the non-standard types. Using these schemas, the rest will be derived automatically.

Sealed traits / coproducts

Tapir supports schema generation for coproduct types (sealed trait hierarchies) of the box, but they need to be defined by hand (as implicit values). To properly reflect the schema in OpenAPI documentation, a discriminator object can be specified.

For example, given following coproduct:

```
sealed trait Entity{
  def kind: String
}
case class Person(firstName:String, lastName:String) extends Entity {
  def kind: String = "person"
}
case class Organization(name: String) extends Entity {
  def kind: String = "org"
}
```

The schema may look like this:

```
val sPerson = implicitly[Schema[Person]]
val sOrganization = implicitly[Schema[Organization]]
implicit val sEntity: Schema[Entity] =
    Schema.oneOf[Entity, String](_.kind, _.toString)("person" -> sPerson, "org" ->␣
↪sOrganization)
```

4.8.4 Customising derived schemas

In some cases, it might be desirable to customise the derived schemas, e.g. to add a description to a particular field of a case class. This can be done by looking up an implicit instance of the `Derived[Schema[T]]` type, and assigning it to an implicit schema. When such an implicit `Schema[T]` is in scope will have higher priority than the built-in low-priority conversion from `Derived[Schema[T]]` to `Schema[T]`.

Schemas for products/coproducts (case classes and case class families) can be traversed and modified using `.modify` method. To traverse collections, use `.each`.

For example:

```
case class Basket(fruits: List[FruitAmount])
case class FruitAmount(fruit: String, amount: Int)
implicit val customBasketSchema: Schema[Basket] = implicitly[Derived[Schema[Basket]]].
↪value
      .modify(_.fruits.each.amount)(_.description("How many fruits?"))
```

There is also an unsafe variant of this method, but it should be avoided in most cases. The "unsafe" prefix comes from the fact that the method takes a list of strings, which represent fields, and the correctness of this specification is not checked.

Non-standard collections can be unwrapped in the modification path by providing an implicit value of `ModifyFunctor`.

4.8.5 Next

Read on about validation.

4.9 Validation

Tapir supports validation for primitive/base types. Validation of composite values, whole data structures, business rules enforcement etc. should be done as part of the server logic of the endpoint, using the dedicated error output (the `E` in `Endpoint[I, E, O, S]`) to report errors.

4.9.1 Single type validation

Validation rules are part of the codec for a given type. They can be specified when creating the codec (using the `Codec.validate()` method):

```
case class MyId(id: String)

implicit val myIdCodec: Codec[String, MyId, TextPlain] = Codec.string
  .mapDecode(decode)(encode)
  .validate(Validator.pattern("^[A-Z].*").contramap(_.id))
```

Validators can also be added to individual inputs/outputs, in addition to whatever the codec provides:

```
val e = endpoint.in(
  query[Int]("amount")
    .validate(Validator.min(0))
    .validate(Validator.max(100)))
```

Validation rules added using the built-in validators are translated to OpenAPI documentation.

Validation rules and automatic codec derivation

When a codec is automatically derived for a type (see custom types), validators for all types (for json this is a recursive process) are looked up through implicit `Validator[T]` values.

Note: Implicit `Validator[T]` values are used *only* when automatically deriving codecs. They are not used when the codec is defined by hand.

Note that to validate a nested member of a case class, it needs to have a unique type (that is, not an `Int`, as providing an implicit `Validator[Int]` would validate all ints in the hierarchy), as validator lookup is type-driven.

To introduce unique types for primitive values, you can use value classes or type tagging.

For example, to support an integer wrapped in a value type in a json body, we need to provide Circe encoders and decoders (if that's the json library that we are using), schema information and a validator:

```
case class Amount(v: Int) extends AnyVal
case class FruitAmount(fruit: String, amount: Amount)

val e: Endpoint[FruitAmount, Unit, Unit, Nothing] = {
  implicit val schemaForAmount: SchemaFor[Amount] = SchemaFor(Schema.SInteger)
  implicit val encoder: Encoder[Amount] = Encoder.encodeInt.contramap(_.v)
  implicit val decode: Decoder[Amount] = Decoder.decodeInt.map(Amount.apply)
  implicit val v: Validator[Amount] = Validator.min(1).contramap(_.v)
  endpoint.in(jsonBody[FruitAmount])
}
```

4.9.2 Decode failures

Codecs support reporting decoding failures, by returning a `DecodeResult` from the `Codec.decode` method. However, this is meant for input/output values which are 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.

To customise error messages that are returned upon validation/decode failures by the server, see error handling.

4.9.3 Enum validators

Validators for enumerations can be created using the `Validator.enum` method, which either:

- takes a type parameter, which should be an abstract, sealed base type, and using a macro determines the possible implementations
- takes the list of possible values

To properly represent possible values in documentation, the enum validator additionally needs an `encode` method, which converts the enum value to a raw type (typically a string). This method is inferred *only* if the validator is directly added to a codec (without any mapping etc.), for example:

```
sealed trait Color
case object Blue extends Color
case object Red extends Color

implicit def plainCodecForColor: PlainCodec[Color] = {
  Codec.string
    .map[Color]((_: String) match {
      case "red"  => Red
      case "blue" => Blue
    })(_.toString.toLowerCase)
    .validate(Validator.enum)
}
```

If the enum is nested within an object, regardless of whether the codec for that object is defined by hand or derived, we need to specify the encode function by hand:

```
implicit def colorValidator: Validator[Color] = Validator.enum.encode(_.toString.
↪toLowerCase)
```

Like other validators, enum validators need to be added to a codec, or through an implicit value, if the codec and validator is automatically derived.

4.9.4 Next

Read on about json support.

4.10 Working with JSON

Json values are supported through codecs, which encode/decode values to json strings. Most often, you'll be using a third-party library to perform the actual json parsing/printing. Currently, Circe, µPickle, Spray JSON and Play JSON are supported.

All of the integrations, when imported into scope, define a `jsonBody[T]` method. This method depends on library-specific implicits being in scope, and derives from them a json codec. The derivation also requires implicit `Schema[T]` and `Validator[T]` instances, which should be automatically derived. For more details see documentation on supporting custom types.

If you have a custom, implicit `Codec[String, T, Json]` instance, you should use the `anyJsonBody[T]` method instead. This description of endpoint input/output, instead of deriving a codec basing on other library-specific implicits, uses the json codec that is in scope.

4.10.1 Circe

To use Circe add the following dependency to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-json-circe" % "0.16.1"
```

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

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

This will allow automatically deriving `Codec`s which, given an in-scope circe `Encoder`/`Decoder` and a `Schema`, 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.

Note that when using Circe's auto derivation, any encoders/decoders for custom types must be in scope as well.

Additionally, the above import brings into scope the `jsonBody[T]` body input/output description, which uses the above codec.

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

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

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

(continues on next page)

(continued from previous page)

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

Circe lets you select an instance of `io.circe.Printer` to configure the way JSON objects are rendered. By default Tapir uses `Printer.nospaces`, which would render:

```
Json.obj(
  "key1" -> Json.fromString("present"),
  "key2" -> Json.Null
)
```

as

```
{"key1":"present","key2":null}
```

Suppose we would instead want to omit `null`-values from the object and pretty-print it. You can configure this by overriding the `jsonPrinter` in `tapir.circe.json.TapirJsonCirce`:

```
object MyTapirJsonCirce extends TapirJsonCirce {
  override def jsonPrinter: Printer = Printer.spaces2.copy(dropNullValues = true)
}

import MyTapirJsonCirce._
```

Now the above JSON object will render as

```
{"key1":"present"}
```

4.10.2 µPickle

To use µPickle add the following dependency to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-json-upickle" % "0.16.1"
```

Next, import the package (or extend the `TapirJsonuPickle` trait, see MyTapir and add `TapirJsonuPickle` not `TapirCirceJson`):

```
import sttp.tapir.json.upickle._
```

µPickle requires a ReadWriter in scope for each type you want to serialize. In order to provide one use the `macroRW` macro in the companion object as follows:

```
import sttp.tapir._
import upickle.default._
import sttp.tapir.json.upickle._

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

object Book {
  implicit val rw: ReadWriter[Book] = macroRW
}

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

Like Circe, µPickle allows you to control the rendered json output. Please see the Custom Configuration of the manual for details.

For more examples, including making a custom encoder/decoder, see TapirJsonuPickleTests.scala

4.10.3 Play JSON

To use Play JSON add the following dependency to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-json-play" % "0.16.1"
```

Next, import the package (or extend the `TapirJsonPlay` trait, see MyTapir and add `TapirJsonPlay` not `TapirCirceJson`):

```
import sttp.tapir.json.play._
```

Play JSON requires `Reads` and `Writes` implicit values in scope for each type you want to serialize.

4.10.4 Spray JSON

To use Spray JSON add the following dependency to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-json-spray" % "0.16.1"
```

Next, import the package (or extend the `TapirJsonSpray` trait, see MyTapir and add `TapirJsonSpray` not `TapirCirceJson`):

```
import sttp.tapir.json.spray._
```

Spray JSON requires a `JsonFormat` implicit value in scope for each type you want to serialize.

4.10.5 Tethys JSON

To use Tethys JSON add the following dependency to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-json-tethys" % "0.16.1"
```

Next, import the package (or extend the `TapirJsonTethys` trait, see MyTapir and add `TapirJsonTethys` not `TapirCirceJson`):

```
import sttp.tapir.json.tethysjson._
```

Tethys JSON requires `JsonReader` and `JsonWriter` implicit values in scope for each type you want to serialize.

4.10.6 Jsoniter Scala

To use Jsoniter-scala add the following dependency to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-jsoniter-scala" % "0.16.1"
```

Next, import the package (or extend the `TapirJsonJsoniter` trait, see MyTapir and add `TapirJsonJsoniter` not `TapirCirceJson`):

```
import sttp.tapir.json.jsoniter._
```

Jsoniter Scala requires `JsonValueCodec` implicit value in scope for each type you want to serialize.

4.10.7 Other JSON libraries

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

4.10.8 Schemas

To derive json codecs automatically, not only implicits from the base library are needed (e.g. a circe `Encoder`/`Decoder`), but also an implicit `SchemaFor[T]` value, which provides a mapping between a type `T` and its schema. A schema-for value contains a single `schema: Schema` field.

See custom types for details.

4.10.9 Next

Read on about working with forms.

4.11 Forms

4.11.1 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.Body[String, Seq[(String, String)]]
formBody[Map[String, String]]: EndpointIO.Body[String, Seq[(String, String)]]
```

Second, form data can be mapped to a case class. The codec for the case class is automatically derived 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`.

4.11.2 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.Body[Seq[RawPart], Seq[AnyPart]]
```

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 mapped 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 `sttp.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]
```

4.11.3 Next

Read on about authentication.

4.12 Authentication

Inputs which carry authentication data wrap another input can be marked as such by declaring them using members of the `auth` object. Apart from predefined codecs for some authentication methods, such inputs will be treated differently when generating documentation. Otherwise, they behave as normal inputs which map to the given type.

Currently, the following authentication inputs are available (assuming `import sttp.tapir._`):

- `auth.apiKey(anotherInput)`: wraps any other input and designates it as an api key. The input is typically a header, cookie or a query parameter
- `auth.basic[T]`: reads data from the `Authorization` header, removing the `Basic` prefix. To parse the data as a base64-encoded username/password combination, use: `basic[UsernamePassword]`.
- `auth.bearer[T]`: reads data from the `Authorization` header, removing the `Bearer` prefix. To get the string as a token, use: `bearer[String]`.
- `auth.oauth2.authorizationCode(authorizationUrl, tokenUrl, scopes, refreshUrl): EndpointInput[String]`: creates an OAuth2 authorization using authorization code - sign in using an auth service (for documentation, requires defining also the `oauth2-redirect.html`, see Generating OpenAPI documentation)

Multiple authentication inputs indicate that all of the given authentication values should be provided. Specifying alternative authentication methods (where only one value out of many needs to be provided) is currently not supported. However, optional authentication can be described by mapping to optional types, e.g. `bearer[Option[String]]`.

When interpreting a route as a server, it is useful to define the authentication input first, to be able to share the authentication logic among multiple endpoints easily. See server logic for more details.

4.12.1 Next

Read on about datatypes integrations.

4.13 Datatypes integrations

4.13.1 Cats datatypes integration

The `tapir-cats` module contains schema, validator and codec instances for some cats datatypes:

```
"com.softwaremill.sttp.tapir" %% "tapir-cats" % "0.16.1"
```

See the `sttp.tapir.codec.cats.TapirCodecCats` trait or `import sttp.tapir.codec.cats._` to bring the implicit values into scope.

4.13.2 Refined integration

If you use refined, the `tapir-refined` module will provide implicit codecs and validators for `T Refined P` as long as a codec for `T` already exists:

```
"com.softwaremill.sttp.tapir" %% "tapir-refined" % "0.16.1"
```

You'll need to extend the `sttp.tapir.codec.refined.TapirCodecRefined` trait or `import sttp.tapir.codec.refined._` to bring the implicit values into scope.

The refined codecs contain a validator which wrap/unwrap the value from/to its refined equivalent.

Some predicates will bind correctly to the vanilla tapir Validator, while others will bind to a custom validator that might not be very clear when reading the generated documentation. Correctly bound predicates can be found in `integration/refined/src/main/scala/sttp/tapir/codec/refined/TapirCodecRefined.scala`. If you are not satisfied with the validator generated by `tapir-refined`, you can provide an implicit `ValidatorForPredicate[T, P]` in scope using 'ValidatorForPredicate.fromPrimitiveValidator' to build it (do not hesitate to contribute your work!).

4.13.3 Enumeratum integration

The `tapir-enumartum` module provides schemas, validators and codecs for Enumeratum enumerations. To use, add the following dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-enumeratum" % "0.16.1"
```

Then, `import sttp.tapir.codec.enumeratum`, or extends the `sttp.tapir.codec.enumeratum.TapirCodecEnumeratum` trait.

This will bring into scope implicit values for values extending `*EnumEntry`.

4.13.4 Enumeration integration

There is no library for the use of the build in scala `Enumeration`, but it can be implemented by hand.

The example code below will generate enums to the open-api documentation.

```
trait EnumHelper { e: Enumeration =>
    import io.circe._

    implicit val enumDecoder: Decoder[e.Value] = Decoder.decodeEnumeration(e)
    implicit val enumEncoder: Encoder[e.Value] = Encoder.encodeEnumeration(e)

    implicit val schemaForEnum: Schema[e.Value] = Schema(SchemaType.SString)
    implicit def validatorForEnum: Validator[e.Value] = Validator.`enum`(e.values.
↪toList, v => Option(v))
}
object Color extends Enumeration with EnumHelper {
    type Color = Value
    val Blue = Value("blue")
    val Red   = Value("red")
}
```

4.14 ZIO integration

The `tapir-zio` module defines type aliases and extension methods which make it more ergonomic to work with ZIO and tapir. Moreover, the `tapir-zio-http4s-server` contains similar extensions useful when exposing the endpoints using the http4s server.

You'll need the following dependencies:

```
"com.softwaremill.sttp.tapir" %% "tapir-zio" % "0.16.1"
"com.softwaremill.sttp.tapir" %% "tapir-zio-http4s-server" % "0.16.1"
```

Next, instead of the usual `import sttp.tapir._`, you should import:

```
import sttp.tapir.ztapir._
```

This brings into scope all of the basic input/output descriptions, which can be used to define an endpoint. Additionally, it defines the `ZEndpoint` type alias, which should be used instead of `Endpoint`.

Note: You should have only one of these imports in your source file. Otherwise, you'll get naming conflicts. The `import sttp.tapir.ztapir._` import is meant as a complete replacement of `import sttp.tapir._`.

4.14.1 Server logic

When defining the business logic for an endpoint, the following methods are available, which replace the *standard ones*:

- `def zServerLogic(logic: I => ZIO[R, E, O]): ZServerEndpoint[R, I, E, O]`
- `def zServerLogicPart(logicPart: T => ZIO[R, E, U])`
- `def zServerLogicForCurrent(logicPart: I => ZIO[R, E, U])`

The first defines complete server logic, while the second and third allow defining server logic in parts.

4.14.2 Exposing endpoints using the http4s server

To bring into scope the extension methods used to interpret a `ZServerEndpoint` as a http4s server, add the following import:

```
import sttp.tapir.server.http4s.ztapir._
```

This adds the following methods on `ZEndpoint`:

- `def toRoutes(logic: I => ZIO[Any, E, O]): HttpRoutes[Task]`
- `def toRoutesR(logic: I => ZIO[R, E, O]): URIO[R, HttpRoutes[Task]]` (when the logic requires an environment)

And the following methods on `ZServerEndpoint` or `List[ZServerEndpoint]`:

- `def toRoutes: HttpRoutes[Task]`
- `def toRoutesR: URIO[R, HttpRoutes[Task]]` (when the logic requires an environment)

4.14.3 Example

Two examples of using the ZIO integration are available. The first showcases basic functionality, while the second shows how to use partial server logic methods:

- ZIO basic example
- ZIO partial server logic example

4.15 Running as an akka-http server

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

```
"com.softwaremill.sttp.tapir" %% "tapir-akka-http-server" % "0.16.1"
```

This will transitively pull some Akka modules in version 2.6. If you want to force your own Akka version (for example 2.5), use sbt exclusion. Mind the Scala version in artifact name:

```
"com.softwaremill.sttp.tapir" %% "tapir-akka-http-server" % "0.16.1" exclude("com.
↪typesafe.akka", "akka-stream_2.12")
```

Now import the package:

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

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

```
I => Future[Either[E, O]]
```

The third recovers errors from failed futures, and hence requires that `E` is a subclass of `Throwable` (an exception); it expects a function of type `I => Future[O]`.

For example:

```
import sttp.tapir._
import sttp.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)
```

Note that these functions take one argument, which is a tuple of type `I`. This means that functions which take multiple arguments need to be converted to a function using a single argument using `.tupled`:

```
def logic(s: String, i: Int): Future[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Nothing] = ???
val aRoute: Route = anEndpoint.toRoute((logic _).tupled)
```

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␣
↪*/)
  }
}
```

4.15.1 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)`.

4.15.2 Configuration

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

4.15.3 Defining an endpoint together with the server logic

It's also possible to define an endpoint together with the server logic in a single, more concise step. See server logic for details.

4.16 Running as an http4s server

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

```
"com.softwaremill.sttp.tapir" %% "tapir-http4s-server" % "0.16.1"
```

and import the package:

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

This adds two extension methods to the `Endpoint` type: `toRoutes` and `toRoutesRecoverErrors`. This first requires the logic of the endpoint to be given as a function of type:

```
I => F[Either[E, O]]
```

where `F[_]` is the chosen effect type. The second recovers errors from failed effects, and hence requires that `E` is a subclass of `Throwable` (an exception); it expects a function of type `I => F[O]`. For example:

```
import sttp.tapir._
import sttp.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 _)
```

Note that these functions take one argument, which is a tuple of type `I`. This means that functions which take multiple arguments need to be converted to a function using a single argument using `.tupled`:

```
def logic(s: String, i: Int): IO[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Nothing] = ???
val aRoute: Route = anEndpoint.toRoute((logic _).tupled)
```

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.

4.16.1 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)`.

4.16.2 Configuration

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

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

4.16.3 Defining an endpoint together with the server logic

It's also possible to define an endpoint together with the server logic in a single, more concise step. See server logic for details.

4.17 Running as a Finatra server

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

```
"com.softwaremill.sttp.tapir" %% "tapir-finatra-server" % "0.16.1"
```

and import the package:

```
import sttp.tapir.server.finatra._
```

or if you would like to use cats-effect project, you can add the following dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-finatra-server-cats" % "0.16.1"
```

and import the packate:

```
import sttp.tapir.server.finatra.cats._
```

This adds extension methods to the `Endpoint` type: `toRoute` and `toRouteRecoverErrors`. The first one requires the logic of the endpoint to be given as a function of type (note this is a Twitter `Future` or a cats-effect project's `Effect` type):

```
I => Future[Either[E, O]]
```

or

An effect type that supports cats-effect's `Effect[F]` type, for example, project Catbird's `Rerunnbale`.

```
I => Rerunnable[Either[E, O]]
```

The second recovers errors from failed futures, and hence requires that `E` is a subclass of `Throwable` (an exception); it expects a function of type `I => Future[O]` or type `I => F[O]` if `F` supports cat-effect's `Effect` type.

For example:

```
import sttp.tapir._
import sttp.tapir.server.finatra._
import com.twitter.util.Future

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

val countCharactersEndpoint: Endpoint[String, Unit, Int, Nothing] =
```

(continues on next page)

(continued from previous page)

```
  endpoint.in(stringBody).out(plainBody[Int])

val countCharactersRoute: FinatraRoute = countCharactersEndpoint.
↪toRoute(countCharacters)
```

or a cats-effect's example:

```
import cats.effect.IO
import sttp.tapir._
import sttp.tapir.server.finatra.cats._

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 countCharactersRoute: FinatraRoute = countCharactersEndpoint.
↪toRoute(countCharacters)
```

Note that these functions take one argument, which is a tuple of type `I`. This means that functions which take multiple arguments need to be converted to a function using a single argument using `.tupled`:

```
def logic(s: String, i: Int): Future[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Nothing] = ???
val aRoute: FinatraRoute = anEndpoint.toRoute((logic _).tupled)
```

Now that you've created the `FinatraRoute`, add `TapirController` as a trait to your `Controller`. You can then add the created route with `addTapirRoute`.

```
class MyController extends Controller with TapirController {
  addTapirRoute(endpoint.toRoute { (s: String, i: Int) => ??? })
}
```

4.17.1 Defining an endpoint together with the server logic

It's also possible to define an endpoint together with the server logic in a single, more concise step. See server logic for details.

4.18 Running as a Play server

To expose endpoint as a play-server first add the following dependencies:

```
"com.softwaremill.sttp.tapir" %% "tapir-play-server" % "0.16.1"
```

and

```
"com.typesafe.play" %% "play-akka-http-server" % "2.8.1"
```

or

```
"com.typesafe.play" %% "play-netty-http-server" % "2.8.1"
```

depending on whether you want to use netty or akka based http-server under the hood.

Then import the package:

```
import sttp.tapir.server.play._
```

This adds two extension methods to the `Endpoint` type: `toRoutes` and `toRoutesRecoverErrors`. This first requires the logic of the endpoint to be given as a function of type:

```
I => Future[Either[E, O]]
```

The second recovers errors from failed effects, and hence requires that `E` is a subclass of `Throwable` (an exception); it expects a function of type `I => Future[O]`. For example:

```
import sttp.tapir._
import sttp.tapir.server.play._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import play.api.routing.Router.Routes

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

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

Note that these functions take one argument, which is a tuple of type `I`. This means that functions which take multiple arguments need to be converted to a function using a single argument using `.tupled`:

```
def logic(s: String, i: Int): Future[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Nothing] = ???
val aRoute: Routes = anEndpoint.toRoute((logic _).tupled)
```

In practice, the routes are put in a class taking an `endpoints.play.server.PlayComponents` parameter. An HTTP server can then be started as in the following example:

```
object Main {
  // JVM entry point that starts the HTTP server
  def main(args: Array[String]): Unit = {
    val playConfig = ServerConfig(port =
      sys.props.get("http.port").map(_.toInt).orElse(Some(9000))
    )
    NettyServer.fromRouterWithComponents(playConfig) { components =>
      val playComponents = PlayComponents.fromBuiltInComponents(components)
      new CounterServer(playComponents).routes orElse new DocumentationServer(
        playComponents
      ).routes
    }
  }
}
```

4.18.1 Configuration

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

4.18.2 Defining an endpoint together with the server logic

It's also possible to define an endpoint together with the server logic in a single, more concise step. See server logic for details.

4.19 Running as a Vert.X server

Endpoints can be mounted as Vert.x `Route`s on top of a Vert.x `Router`.

Use the following dependency

```
"com.softwaremill.sttp.tapir" %% "tapir-vertx-server" % "0.16.1"
```

Then import the package:

```
import sttp.tapir.server.vertx._
```

This adds extension methods to the `Endpoint` type:

- `route(logic: I => Future[Either[E, O]])`: returns a function `Router => Route` that will create a route matching the endpoint definition, and attach your `logic` as an handler. Errors will be recovered automatically (but generically)
- `routeRecoverErrors(logic: I => Future[O])`: returns a function `Router => Route` that will create a route matching the endpoint definition, and attach your `logic` as an handler. You're providing your own way to deal with errors happening in the `logic` function.
- `blockingRoute(logic: I => Future[Either[E, O]])`: returns a function `Router => Route` that will create a route matching the endpoint definition, and attach your `logic` as an blocking handler. Errors will be recovered automatically (but generically)
- `blockingRouteRecoverErrors(logic: I => Future[O])`: returns a function `Router => Route` that will create a route matching the endpoint definition, and attach your `logic` as a blocking handler. You're providing your own way to deal with errors happening in the `logic` function.

The methods recovering errors from failed effects, require `E` to be a subclass of `Throwable` (an exception); and expect a function of type `I => Future[O]`. For example:

Note that these functions take one argument, which is a tuple of type `I`. This means that functions which take multiple arguments need to be converted to a function using a single argument using `.tupled`:

```
def logic(s: String, i: Int): Future[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Nothing] = ???
val aRoute: Router => Route = anEndpoint.route(router, (logic _).tupled)
```

In practice, routes will be mounted on a router, this router can then be used as a request handler for your http server. An HTTP server can then be started as in the following example:

```
object Main {
  // JVM entry point that starts the HTTP server
  def main(args: Array[String]): Unit = {
    val vertx = Vertx.vertx()
    val server = vertx.createHttpServer()
    val router = Router.router(vertx)
    val anEndpoint: Endpoint[(String, Int), Unit, String, Nothing] = ??? // your␣
↪definition here
    def logic(s: String, i: Int): Future[Either[Unit, String]] = ??? // your logic␣
↪here
```

(continues on next page)

(continued from previous page)

```
    anEndpoint.route(router, (logic _).tupled) // your endpoint is now attached to␣
↪the router, and the route has been created
    server.requestHandler(router).listenFuture(9000)
  }
}
```

4.19.1 Configuration

Every endpoint can be configured by providing an implicit `VertxEndpointOptions`, see server options for details. You can also provide your own `ExecutionContext` to execute the logic.

4.19.2 Defining an endpoint together with the server logic

It's also possible to define an endpoint together with the server logic in a single, more concise step. See server logic for details.

4.20 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 (see also error handling)
- debug-logging of request handling

To customise the server options, define an implicit value, which will be visible when converting an endpoint or multiple endpoints to a route/routes. For example, for `AkkaHttpServerOptions`:

```
implicit val customServerOptions: AkkaHttpServerOptions =
  AkkaHttpServerOptions.default.copy(...)
```

4.21 Server logic

The logic that should be run when an endpoint is invoked can be passed when interpreting the endpoint as a server, and converting to a server-implementation-specific route. However, there's also the option to define an endpoint coupled with the server logic - either given entirely or gradually, in parts. This might make it easier to work with endpoints and their collections in a server setting.

4.21.1 Defining an endpoint together with the server logic

It's possible to combine an endpoint description with the server logic in a single object, `ServerEndpoint[I, E, O, S, F]`. Such an endpoint contains not only an endpoint of type `Endpoint[I, E, O, S]`, but also a logic function `I => F[Either[E, O]]`, for some effect `F`.

The book example can be more concisely written as follows:

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

val countCharactersServerEndpoint: ServerEndpoint[String, Unit, Int, Nothing, Future]␣
↪=
  endpoint.in(stringBody).out(plainBody[Int]).serverLogic { s =>
    Future.successful(Right[Unit, Int](s.length))
  }

val countCharactersRoute: Route = countCharactersServerEndpoint.toRoute
```

A `ServerEndpoint` can then be converted to a route using `.toRoute`/`.toRoutes` methods (without any additional parameters; the exact method name depends on the server interpreter), or to documentation.

Moreover, a list of server endpoints can be converted to routes or documentation as well:

```
val endpoint1 = endpoint.in("hello").out(stringBody)
  .serverLogic { _ => Future.successful(Right("world")) }

val endpoint2 = endpoint.in("ping").out(stringBody)
  .serverLogic { _ => Future.successful(Right("pong")) }

val route: Route = List(endpoint1, endpoint2).toRoute
```

Note that when dealing with endpoints which have multiple input parameters, the server logic function is a function of a *single* argument, which is a tuple; hence you'll need to pattern-match using `case` to extract the parameters:

```
val echoEndpoint = endpoint
  .in(query[Int]("count"))
  .in(stringBody)
  .out(stringBody)
  .serverLogic { case (count, body) =>
     Future.successful(Right(body * count))
  }
```

Recovering errors from failed effects

If your error type is an exception (extends `Throwable`), and if errors that occur in the server logic are represented as failed effects, you can use a variant of the methods above, which extract the error from the failed effect, and respond with the error output appropriately.

This can be done with the `serverLogicRecoverErrors(f: I => F[O])` method. Note that the `E` type parameter isn't directly present here; however, the method also contains a requirement that `E` is an exception, and will only recover errors which are subtypes of `E`. Any others will be propagated without changes.

For example:

```
case class MyError(msg: String) extends Exception
val testEndpoint = endpoint
  .in(query[Boolean]("fail"))
  .errorOut(stringBody.map(MyError)(_.msg))
  .out(stringBody)
  .serverLogicRecoverErrors { fail =>
    if (fail) {
```

(continues on next page)

(continued from previous page)

```
      Future.successful("OK") // note: no Right() wrapper
    } else {
      Future.failed(new MyError("Not OK")) // no Left() wrapper, a failed future
    }
  }
```

4.21.2 Extracting common route logic

Quite often, especially for authentication, some part of the route logic is shared among multiple endpoints. However, these functions don't compose in a straightforward way, as authentication usually operates on a single input, which is only a part of the whole logic's input. That's why there are two options for providing the server logic in parts.

Defining an extendable, base endpoint with partial server logic

First, you might want to define a base endpoint, with some part of the server logic already defined, and later extend it with additional inputs/outputs, and successive logic parts.

When using this method, note that the error type must be fully defined upfront, and cannot be extended later. That's because all of the partial logic functions can return either an error (of the given type), or a partial result.

To define partial logic for the inputs defined so far, you should use the `serverLogicForCurrent` method on an endpoint. This accepts a method, `f: I => F[Either[E, U]]`, which given the entire (current) input defined so far, returns either an error, or a partial result.

For example, we can create a partial server endpoint given an authentication function, and an endpoint describing the authentication inputs:

```
case class User(name: String)
def auth(token: String): Future[Either[Int, User]] = Future {
  if (token == "secret") Right(User("Spock"))
  else Left(1001) // error code
}

val secureEndpoint: PartialServerEndpoint[User, Unit, Int, Unit, Nothing, Future] =␣
↪endpoint
  .in(header[String]("X-AUTH-TOKEN"))
  .errorOut(plainBody[Int])
  .serverLogicForCurrent(auth)
```

The result is a value of type `PartialServerEndpoint`, which can be extended with further inputs and outputs, just as a normal endpoint (except for error outputs, which are fixed).

Successive logic parts (again consuming the entire input defined so far, and producing another partial result) can be given by calling `serverLogicForCurrent` again. The logic can be completed - given a tuple of partial results, and unconsumed inputs - using the `serverLogic` method:

```
val secureHelloWorld1WithLogic = secureEndpoint.get
  .in("hello1")
  .in(query[String]("salutation"))
  .out(stringBody)
  .serverLogic { case (user, salutation) => Future(Right(s"$salutation, ${user.name}!
↪")) }
```

Once the server logic is complete, a `ServerEndpoint` is returned, which can be then interpreted as a server.

The methods mentioned also have variants which recover errors from failed effects (as described above), using `serverLogicForCurrentRecoverErrors` and `serverLogicRecoverErrors`.

Providing server logic in parts, for an already defined endpoint

Secondly, you might already have the entire endpoint defined upfront (e.g. in a separate module, which doesn't know anything about the server logic), and would like to provide the server logic in parts.

This can be done using the `serverLogicPart` method, which takes a function of type `f: T => F[Either[E, U]]` as a parameter. Here `T` is some prefix of the endpoint's input `I`. The function then consumes some part of the input, and produces an error, or a partial result.

Unlike previously, here the endpoint is considered complete, and cannot be later extended: no additional inputs or outputs can be defined.

Note: Note that the partial logic functions should be fully typed, to help with inference. It might not be possible for the compiler to infer, which part of the input should be consumed.

For example, if we have an endpoint:

```
val secureHelloWorld2: Endpoint[(String, String), Int, String, Nothing] = endpoint
  .in(header[String]("X-AUTH-TOKEN"))
  .errorOut(plainBody[Int])
  .get
  .in("hello2")
  .in(query[String]("salutation"))
  .out(stringBody)
```

We can provide the server logic in parts (using the same `auth` method as above):

```
val secureHelloWorld2WithLogic = secureHelloWorld2
  .serverLogicPart(auth)
  .andThen { case (user, salutation) => Future(Right(s"$salutation, ${user.name}!")) }
```

Here, we first define a single part using `serverLogicPart`, and then complete the logic (consuming the remaining inputs) using `andThen`. The result is, like previously, a `ServerEndpoint`, which can be interpreted as a server.

Multiple parts can be provided using `andThenPart` invocations (consuming successive input parts). There are also variants of the methods, which recover errors from failed effects: `serverLogicPartRecoverErrors`, `andThenRecoverErrors` and `andThenPartRecoverErrors`.

4.21.3 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 by specifying how an output maps to the status code.

4.22 Error handling

4.22.1 Exception handling

There's no exception handling built into tapir. However, tapir contains a more general error handling mechanism, as the endpoints can contain dedicated error outputs.

If the logic function, which is passed to the server interpreter, fails (i.e. throws an exception, which results in a failed `Future` or `IO`/`Task`), this is propagated to the library (akka-http or http4s).

However, any exceptions can be recovered from and mapped to an error value. For example:

```
type ErrorInfo = String

def logic(s: String): Future[Int] = ...

def handleErrors[T](f: Future[T]): Future[Either[ErrorInfo, T]] =
  f.transform {
    case Success(v) => Success(Right(v))
    case Failure(e) =>
      logger.error("Exception when running endpoint logic", e)
      Success(Left(e.getMessage))
  }

endpoint
  .errorOut(plainBody[ErrorInfo])
  .out(plainBody[Int])
  .in(query[String]("name"))
  .toRoute((logic _).andThen(handleErrors))
```

In the above example, errors are represented as `String`s (aliased to `ErrorInfo` for readability). When the logic completes successfully an `Int` is returned. Any exceptions that are raised are logged, and represented as a value of type `ErrorInfo`.

Following the convention, the left side of the `Either[ErrorInfo, T]` represents an error, and the right side success.

Alternatively, errors can be recovered from failed effects and mapped to the error output - provided that the `E` type in the endpoint description is itself a subclass of exception. This can be done using the `toRouteRecoverErrors` method.

4.22.2 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. The path doesn't match if a path segment is missing, there's a constant value mismatch or a decoding error (e.g. parsing a segment to an `Int` fails)
- 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 causes any decode failure, or if the decoding a path capture causes a validation error.

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" 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.

Note that the decode failure handler is used **only** for failures that occur during decoding of path, query, body and header parameters - while invoking `Codec.decode`. It does not handle any failures or exceptions that occur when invoking the logic of the endpoint.

Default failure handler

The default decode failure handler is a case class, consisting of functions which decide whether to respond with an error or return a "no match", create error messages and create the response.

To reuse the existing default logic, parts of the default behavior can be swapped, e.g. to return responses in a different format (other than textual):

```
case class MyFailure(msg: String)
def myFailureResponse(statusCode: StatusCode, message: String): DecodeFailureHandling␣
↪=
  DecodeFailureHandling.response(statusCode.and(jsonBody[MyFailure]))(
    (statusCode, MyFailure(message))
  )

val myDecodeFailureHandler = ServerDefaults.decodeFailureHandler.copy(
  response = myFailureResponse
)

implicit val myServerOptions: AkkaHttpServerOptions = AkkaHttpServerOptions.default.
↪copy(
  decodeFailureHandler = myDecodeFailureHandler
)
```

Note that when specifying that a response should be returned upon a failure, we need to provide the endpoint output which should be used to create the response, as well as a value for this output.

The default decode failure handler also has the option to return a `400 Bad Request`, instead of a no-match (ultimately leading to a `404 Not Found`), when the "shape" of the path matches (that is, the number of segments in the request and endpoint's paths are the same), but when decoding some part of the path ends in an error. See the `badRequestOnPathErrorIfPathShapeMatches` in `ServerDefaults`.

Finally, you can provide custom error messages for validation errors (which optionally describe what failed) and failure errors (which describe the source of the error).

A completely custom implementation of the `DecodeFailureHandler` function can also be used.

4.23 Debugging

When dealing with multiple endpoints, how to find out which endpoint handled a request, or why an endpoint didn't handle a request?

For this purpose, tapir provides optional logging. The logging options (and messages) can be customised by changing the default `LogRequestHandling` class, which is part of server options.

The following can be customised:

1. `logWhenHandled`: when a request is handled by an endpoint, or when the inputs can't be decoded, and the decode failure maps to a response (default: `true`, `DEBUG` log)
2. `logAllDecodeFailures`: when the inputs can't be decoded, and the decode failure doesn't map to a response (the next endpoint will be tried; default: `false`, `DEBUG` log)
3. `logLogicExceptions`: when there's an exception during evaluation of the server logic (default: `true`, `ERROR` log)

Logging all decode failures (2) might be helpful when debugging, but can also produce a large amount of logs, hence it's disabled by default.

Even if logging for a particular category (as described above) is set to `true`, normal logger rules apply - if you don't see the logs, please verify your logging levels for the appropriate packages.

4.24 Using as an sttp client

Add the dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-sttp-client" % "0.16.1"
```

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

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

This adds the two extension methods to any `Endpoint`:

- `toSttpRequestUnsafe(Uri)`: given the base URI returns a function, which might throw an exception if decoding of the result fails

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

- `toSttpRequest(Uri)`: given the base URI returns a function, which represents decoding errors as the `DecodeResult` class

```
I => Request[DecodeResult[Either[E, O]], Nothing]
```

Note that these are a one-argument functions, where the single argument is the input of end endpoint. This might be a single type, a tuple, or a case class, depending on the endpoint description.

After providing the input parameters, a description of the request to be made is returned, with the input value encoded as appropriate request parameters: path, query, headers and body. This can be further customised and sent using any sttp backend. The response will then contain the decoded error or success values (note that this can be the body enriched with data from headers/status code).

See the runnable example for example usage.

4.25 Generating OpenAPI documentation

To use, add the following dependencies:

```
"com.softwaremill.sttp.tapir" %% "tapir-openapi-docs" % "0.16.1"
"com.softwaremill.sttp.tapir" %% "tapir-openapi-circe-yaml" % "0.16.1"
```

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 sttp.tapir.openapi.OpenAPI
import sttp.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 documentation is generated in a large part basing on schemas. Schemas can be automatically derived and customised.

Quite often, you'll need to define the servers, through which the API can be reached. To do this, you can modify the returned `OpenAPI` case class either directly or by using a helper method:

```
val docs: OpenAPI = booksListing.toOpenAPI("My Bookshop", "1.0")
  .servers(List(Server("https://api.example.com/v1").description("Production server
↪")))
```

Multiple endpoints can be converted to an `OpenAPI` instance by calling the extension method on a list of endpoints:

```
List(addBook, booksListing, booksListingByGenre).toOpenAPI("My Bookshop", "1.0")
```

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

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

println(docs.toYaml)
```

4.25.1 Options

Options can be customised by providing an implicit instance of `OpenAPIDocsOptions`, when calling `.toOpenAPI`.

- `operationIdGenerator`: each endpoint corresponds to an operation in the OpenAPI format and should have a unique operation id. By default, the `name` of endpoint is used as the operation id, and if this is not available, the operation id is auto-generated by concatenating (using camel-case) the request method and path.

4.25.2 Exposing OpenAPI documentation

Exposing the OpenAPI documentation can be very application-specific. However, tapir contains modules which contain akka-http/http4s routes for exposing documentation using Swagger UI or Redoc:

```
"com.softwaremill.sttp.tapir" %% "tapir-swagger-ui-akka-http" % "0.16.1"
"com.softwaremill.sttp.tapir" %% "tapir-redoc-akka-http" % "0.16.1"
"com.softwaremill.sttp.tapir" %% "tapir-swagger-ui-http4s" % "0.16.1"
"com.softwaremill.sttp.tapir" %% "tapir-redoc-http4s" % "0.16.1"
```

Note: `tapir-swagger-ui-akka-http` transitively pulls some Akka modules in version 2.6. If you want to force your own Akka version (for example 2.5), use sbt exclusion. Mind the Scala version in artifact name:

```
"com.softwaremill.sttp.tapir" %% "tapir-swagger-ui-akka-http" % "0.16.1" exclude("com.
↪typesafe.akka", "akka-stream_2.12")
```

Usage example for akka-http:

```
import sttp.tapir.docs.openapi._
import sttp.tapir.openapi.circe.yaml._
import sttp.tapir.swagger.akkahttp.SwaggerAkka

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

(continues on next page)

(continued from previous page)

```
// add to your akka routes
new SwaggerAkka(docsAsYaml).routes
```

For redoc, use `RedocAkkaHttp`.

For http4s, use the `SwaggerHttp4s` or `RedocHttp4s` classes.

4.26 Testing

4.26.1 White box testing

If you are unit testing your application you should stub all external services.

If you are using sttp client to send HTTP requests, and if the externals apis, which yours application consumes, are described using tapir, you can create a stub of the service by converting endpoints to `SttpBackendStub` (see the sttp documentation for details on how the stub works).

Add the dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-sttp-stub-server" % "0.16.1"
```

And the following import:

```
import sttp.tapir.server.stub._
```

Then, convert any endpoint to `SttpBackendStub`:

```
val endpoint = sttp.tapir.endpoint
  .in("api" / "sometest4")
  .in(query[Int]("amount"))
  .post
  .out(jsonBody[ResponseWrapper])

implicit val backend = SttpBackendStub
  .apply(idMonad)
  .whenRequestMatchesEndpoint(endpoint)
  .thenSuccess(ResponseWrapper(1.0))
```

4.26.2 Black box testing

When testing application as a whole component, running for example in docker, you might want to stub external services with which you application interacts, to steer their behaviors.

To do that you might want to use well-known solutions like e.g. wiremock or mock-server, but if their api is described using tapir you might want to use livestub, which combines nicely with the rest of the sttp ecosystem.

4.27 Other interpreters

At its core, tapir creates a data structure describing the HTTP endpoints. This data structure can be freely interpreted also by code not included in the library. Below is a list of projects, which provide tapir interpreters.

4.27.1 GraphQL

Caliban allows you to easily turn your Tapir endpoints into a GraphQL API. More details in the documentation.

4.27.2 tapir-gen

tapir-gen extends tapir to do client code generation. The goal is to auto-generate clients in multiple-languages with multiple libraries.

scala-opentracing contains a module which provides a small integration layer that allows you to create traced http endpoints from tapir Endpoint definitions.

4.28 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 TapirAkkaHttpServer
  with TapirSttpClient
  with TapirJsonCirce
  with TapirOpenAPICirceYaml
  with TapirAliases
```

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

4.29 Troubleshooting

4.29.1 StackOverflowException during compilation

Sidenote for scala 2.12.4 and higher: if you encounter an issue with compiling your project because of a `StackOverflowException` related to this scala bug, please increase your stack memory. Example:

```
sbt -J-Xss4M clean compile
```

4.29.2 Logging of generated macros code

For some cases, it may be helpful to examine how generated macros code looks like. To do that, just set an environmental variable and check compilation logs for details.

```
export TAPIR_LOG_GENERATED_CODE=true
```

4.30 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.

4.30.1 Acknowledgments

Tuple-concatenating code is copied from akka-http

Generic derivation configuration is copied from circe