tapir documentation

Release 0.x

Adam Warski

Feb 22, 2023

Contents

1 Code teaser 3

2 Contents 5

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]`
- 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 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")
```

CHAPTER 2

Contents

2.1 Quickstart

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

```
"com.softwaremill.tapir" %% "tapir-core" % "0.11.1"
```

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 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 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
```

2.2 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
- Books example
- ZIO example, using http4s

2.2.1 Other examples

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

2.2.2 Blogs, articles

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

2.3 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

2.3.1 Similar projects

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

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

2.4 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`.

2.4.1 Infallible endpoints

Note that the empty `endpoint` description maps no values to either error and success outputs, however errors are still represented and allow 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.

2.4.2 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.

2.4.3 Next

Read on about describing endpoint inputs/outputs.

2.5 Defining endpoint's input/output

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 `Seq[String]`
- `query[T](name)` captures a query parameter with the given name
- `queryParams` captures all query parameters, represented as `MultiQueryParams`
- `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 `Seq[(String, String)]`
- `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]`
- `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:

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

2.5.1 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, descriptions can be combined using the `.and` method. Such a combination results in an input/output, which maps to a tuple of the given types, and 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.

2.5.2 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, 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))
```

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.

> Note that this kind of mapping is only meant for isomorphic transformations and grouping inputs/outputs into custom types. To support custom types, where one of the transformations might fail, see codecs and validation.

2.5.3 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)`".

2.5.4 Next

Read on about status codes.

2.6 Status codes

To provide a (varying) status code of a server response, use the `statusCode` output, which maps to a value of type `type tapir.model.StatusCode` (which is an alias for `Int`). The `tapir.model.StatusCodes` object contains known status codes as constants. This type of output is used only when interpreting the endpoint as a server.

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

2.6.1 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
```

(continues on next page)

(continued from previous page)

```
case class Unknown(code: Int, msg: String) extends ErrorInfo
case object NoContent extends ErrorInfo

val baseEndpoint = endpoint.errorOut(
  oneOf(
    statusMapping(StatusCodes.NotFound, jsonBody[NotFound].description("not found")),
    statusMapping(StatusCodes.Unauthorized, jsonBody[Unauthorized].description(
↪"unauthorized")),
    statusMapping(StatusCodes.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.

2.6.2 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.

2.6.3 Next

Read on about codecs.

2.7 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).

2.7.1 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.

Note that only textual bodies can be optional (optional binary/streaming bodies aren't supported). That's because a body cannot be missing - there's always *some* body. This is unlike e.g. a query parameter: for which a value can be present, a value can be empty (but defined!), or the parameter might be missing altogether - which corresponds to a `None`. That's why only strict (non-streaming) textual bodies, which are empty (`""`), will be considered as an empty value (`None`), if the codec allows optional values.

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.

2.7.2 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.

The schema is left unchanged when mapping over a codec, as the underlying representation of the value doesn't change.

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

2.7.3 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.

2.7.4 Next

Read on about custom types.

2.8 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.

2.8.1 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
- codec meta-data (`CodecMeta`) consisting of:
  - schema of the type (for documentation)
  - validator for the type
  - media type (`text/plain`, `application/json` etc.)
  - type of the raw value, to which data is serialised (`String`, `Int` 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:

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

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, TextPlain, _] = Codec.stringPlainCodecUtf8
  .mapDecode(decode)(encode)
```

> Note that inputs/outputs can also be mapped over. However, this kind of mapping is always an isomorphism, doesn't allow any validation or reporting decode errors. Hence, it should be used only for grouping inputs or outputs from a tuple into a custom type.

2.8.2 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 `SchemaFor[T]` implicit

Schema derivation

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

2.8.3 Next

Read on about validation.

2.9 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.

2.9.1 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[MyId, TextPlain, _] = Codec.stringPlainCodecUtf8
  .mapDecode(decode)(encode)
  .validate(Validator.pattern("^[A-Z].*").contramap(_.id))
```

Or added to individual inputs/outputs:

```
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 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])
}
```

2.9.2 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.

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.

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

2.9.3 Next

Read on about json support.

2.10 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, µPickle and Play JSON are supported.

2.10.1 Circe

To use Circe add the following dependency to your project:

```
"com.softwaremill.tapir" %% "tapir-json-circe" % "0.11.1"
```

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

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

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

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]
```

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"}
```

2.10.2 µPickle

To use µPickle add the following dependency to your project:

```
"com.softwaremill.tapir" %% "tapir-json-upickle" % "0.11.1"
```

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

```
import 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 tapir._
import upickle.default._
import 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

2.10.3 Play JSON

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

```
"com.softwaremill.tapir" %% "tapir-json-play" % "0.11.1"
```

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

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

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

2.10.4 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).

2.10.5 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.

2.10.6 Next

Read on about working with forms.

2.11 Form support

2.11.1 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[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 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`.

2.11.2 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[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]
```

2.11.3 Next

Read on about authentication.

2.12 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 the given type.

Currently, the following authentication inputs are available (assuming `import 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: EndpointInput[UsernamePassword]`: maps to the base64-encoded username/password pair in the `Authorization` header
- `auth.bearer: EndpointInput[String]`: maps to `Bearer [token]` in the `Authorization` header
- `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.

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.

2.12.1 Next

Read on a summary on implicits for custom types.

2.13 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" % "0.11.1"
```

and import the package:

```
import 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 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)
```

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␣
↪*/)
  }
}
```

2.13.1 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)`.

2.13.2 Configuration

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

2.13.3 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.

2.14 Running as an http4s server

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

```
"com.softwaremill.tapir" %% "tapir-http4s-server" % "0.11.1"
```

and import the package:

```
import 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 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 _)
```

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.

2.14.1 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)`.

2.14.2 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.

2.14.3 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.

2.15 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(...)
```

2.16 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 an server-implementation-specific route. But there are other options available as well, which might make it easier to work with collections of endpoints.

2.16.1 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`.

For example, the book example can be more concisely written as follows:

```
import tapir._
import 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), 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("world") }

val endpoint2 = endpoint.in("ping").out(stringBody)
  .serverLogic { _ => Future.successful("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(body * count)
  }
```

2.16.2 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. Suppose you have the following methods:

```
type AuthToken = String

def authFn(token: AuthToken): Future[Either[ErrorInfo, User]]
def logicFn(user: User, data: String, limit: Int): Future[Either[ErrorInfo, Result]]
```

which you'd like to apply to an endpoint with type:

```
val myEndpoint: Endpoint[(AuthToken, String, Int), ErrorInfo, Result, Nothing] = ...
```

To avoid composing these functions by hand, tapir defines helper extension methods, `andThenFirst` and `andTheFirstE`. The first one should be used when errors are represented as failed wrapper types (e.g. failed futures), the second is errors are represented as `Either`s.

This extension method is defined in the same traits as the route interpreters, both for `Future` (in the akka-http interpreter) and for an arbitrary monad (in the http4s interpreter), so importing the package is sufficient to use it:

```
import tapir.server.akkahttp._
val r: Route = myEndpoint.toRoute((authFn _).andThenFirstE((logicFn _).tupled))
```

Writing down the types, here are the generic signatures when using `andThenFirst` and `andThenFirstE`:

```
f1: T => Future[U]
f2: (U, A1, A2, ...) => Future[O]
(f1 _).andThenFirst(f2): (T, A1, A2, ...) => Future[O]

f1: T => Future[Either[E, U]]
f2: (U, A1, A2, ...) => Future[Either[E, O]]
(f1 _).andThenFirstE(f2): (T, A1, A2, ...) => Future[Either[E, O]]
```

2.16.3 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.

2.17 Error handling

2.17.1 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 {
```

(continues on next page)

(continued from previous page)

```
    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.

2.17.2 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.
- 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" 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 can be adapted to return responses in a different format (other than textual). To reuse the existing logic, but using a different format, create the failure handler as follows:

```
case class MyFailure(msg: String)
def myFailureResponse(statusCode: StatusCode, message: String): DecodeFailureHandling␣
↪=
```

(continues on next page)

(continued from previous page)

```
  DecodeFailureHandling.response(statusCode.and(jsonBody[MyFailure]))(
    (statusCode, MyFailure(message))
  )

val myDecodeFailureHandler = ServerDefaults.decodeFailureHandlerUsingResponse(
  myFailureResponse,
  badRequestOnPathFailureIfPathShapeMatches = false,
  validationErrorToMessage
)
```

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 fails. By default, any kind of decode failure in the path will result in a no-match being returned.

Finally, you can provide custom error messages for validation errors. By default, these messages are appended to the decode failure message (see the `validationErrorToMessage` method in `ServerDefaults`).

2.18 Debugging servers

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 `LoggingOptions` class, which is part of server options.

The following can be logged:

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

By default, logs of type (1) and (3) are logged. Logging all decode failures (2) might be helpful when debugging, but can also produce a large amount of logs.

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.

2.19 Using as an sttp client

Add the dependency:

```
"com.softwaremill.tapir" %% "tapir-sttp-client" % "0.11.1"
```

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

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

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

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

Note that this is a one-argument function, 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. This can be further customised and sent using any sttp backend.

See the runnable example for example usage.

2.20 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.

2.20.1 tapir-gen

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

2.21 Generating OpenAPI documentation

To use, add the following dependencies:

```
"com.softwaremill.tapir" %% "tapir-openapi-docs" % "0.11.1"
"com.softwaremill.tapir" %% "tapir-openapi-circe-yaml" % "0.11.1"
```

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.

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 tapir.openapi.circe.yaml._

println(docs.toYaml)
```

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. This can be customised by providing an implicit instance of `OpenAPIDocsOptions`.

2.21.1 Exposing OpenAPI documentation

Exposing the OpenAPI documentation can be very application-specific. However, tapir contains two modules which contain akka-http/http4s routes for exposing documentation using the swagger ui:

```
"com.softwaremill.tapir" %% "tapir-swagger-ui-akka-http" % "0.11.1"
"com.softwaremill.tapir" %% "tapir-swagger-ui-http4s" % "0.11.1"
```

Usage example for akka-http:

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

val docsAsYaml: String = myEndpoints.toOpenAPI("My App", "1.0").toYaml
// add to your akka routes
new SwaggerAkka(docsAsYaml).routes
```

For http4s, use the `SwaggerHttp4s` class.

2.22 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 TapirCirceJson
  with TapirOpenAPICirceYaml
  with TapirAliases
```

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

2.23 Design notes

The shape of tapir's inputs and outputs is fixed and in some ways constrained. Below you'll find some motivation behind this design, as well as alternatives.

The *input* of an endpoint is always a product of values: that is, each new input extends the list of values that the endpoint's input maps to. A new input can contribute 0 values (in case of fixed paths), 1 value (query parameter, path capture, ...), or many values (a composite input).

What is *not* possible, is describing a *coproduct*: specifying, that the input is either a value of one type, or a value of another type. Coproducts are harder than products as they need a *discriminator*: some kind of value, basing on which it can be decided, which of the input alternatives to choose.

It would be possible to extend tapir and allow coproducts, at the expense of complicating the API. However, use-cases which require such mappings are rare (or so it seems), so this isn't currently implemented. And there's always a "back door": an alternative of values can be described as a "flattened" product of optional values, with additional input validation.

Outputs are a bit more complicated. First of all, success and error outputs are separate. This defines a top-level coproduct, where the output is either mapped to the branch error, or the success branch. The discriminator in this case is the status code.

However, both error and success outputs can contain coproducts as well, using the `oneOf` output. The `oneOf` output specifies a number of alternative outputs, again discriminated using fixed status code values (which is important to be able to generate documentation). The types, to which the branches should map, have to form an inheritance hierarchy, e.g.:

```
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

val baseEndpoint = endpoint.errorOut(
  oneOf(
    statusMapping(StatusCodes.NotFound, jsonBody[NotFound].description("not found")),
    statusMapping(StatusCodes.Unauthorized, jsonBody[Unauthorized]),
    statusDefaultMapping(jsonBody[Unknown].description("unknown"))
  )
)
```

Again, this could be generalised to allow other discriminators (e.g. on fixed header values), however there are no compelling use-cases which would justify this.

It would also be possible to generalise the error/success outputs into a single output type. Users could then use the `oneOf` output to differentiate between errors and successes, for example:

```
val e1: Endpoint[Unit, Either[String, Book], Nothing] = endpoint
  .out(either(
    statusCode(200) -> jsonBody[Book],
    statusCode(400) -> stringBody
  ))

val e2: Endpoint[Unit, Either[ErrorInfo, Book], Nothing] = endpoint
  .out(either(
    statusCode(200) -> jsonBody[Book],
    otherwise -> oneOf[ErrorInfo](
      statusCode(404) -> jsonBody[NotFound],
      statusCode(403) -> jsonBody[Unauthorized],
      statusCode(400) -> jsonBody[Unknown]
    )))
```

However, error outputs are almost always *different* from success outputs, so it's worth complicating the API to support this distinction as a first-class construct. Moreover, quite often endpoints share the error output description, while the success output vary from endpoint to endpoint.

2.24 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.

2.24.1 Acknowledgments

Tuple-concatenating code is copied from akka-http

Generic derivation configuration is copied from circe