tapir documentation

Release 0.x

Adam Warski

Sep 14, 2021

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]`
  - *Netty*
  - *Finatra* `http.Controller`
  - *Play* `Route`
  - *ZIO Http* `Http`
  - *aws* through Lambda/SAM/Terraform
- a client, which is a function from input parameters to output parameters. Currently supported:
  - *sttp*
  - *Play*
  - *Http4s*
- documentation. Currently supported:
  - *OpenAPI*
  - *AsyncAPI*

Tapir is licensed under Apache2, the source code is available on 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!

Tapir is available:

- all modules - Scala 2.12 and 2.13 on the JVM
- selected modules (core, http4s server, sttp client, openapi, some js and datatype integrations) - Scala 3 on the JVM
- selected modules (sttp client, some js and datatype integrations) - Scala 2.12 and 2.13 using Scala.JS.

CHAPTER 1

Code teaser

```
import sttp.tapir._
import sttp.tapir.generic.auto._
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)


// Define an endpoint

val booksListing: Endpoint[(BooksFromYear, Limit, AuthToken), String, List[Book],␣
↪Any] =
  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]])


// Generate OpenAPI documentation

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

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


// Convert to akka-http Route
```

(continues on next page)

(continued from previous page)

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

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


// Convert to sttp Request

import sttp.tapir.client.sttp.SttpClientInterpreter
import sttp.client3._

val booksListingRequest: Request[DecodeResult[Either[String, List[Book]]], Any] =
  SttpClientInterpreter()
    .toRequest(booksListing, Some(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" % "0.19.0-M9"
```

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.

Many of tapir functionalities come as builder methods in the main package, hence it's easiest to work with tapir if you import the main package entirely, i.e.:

```
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

- Hello world server, using ZIO http
- 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
- Basic Authentication, using akka-http
- Partial server logic (authentication), using akka-http
- Authorization using Github (OAuth2) with http4s
- Books example
- Books example with semiauto derivation
- ZIO example, using http4s
- ZIO example with environment, using http4s
- ZIO partial server logic example, using http4s
- ZIO example using ZIO http
- Streaming body, using akka-http
- Streaming body, using http4s + fs2
- Web sockets server, using akka-http
- Web sockets client, using akka-http
- Web sockets server, using http4s
- Client, using http4s
- mockserver integration
- OpenAPI extensions
- Prometheus metrics

4.2.1 Other examples

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

For a quick start using http4s and tapir you can use a gitter8 template. A new project can be created using: `sbt new https://codeberg.org/wegtam/http4s-tapir.g8.git`

4.2.2 Blogs, articles

- Designing tapir's WebSockets support
- Three easy endpoints
- tAPIr's Endpoint meets ZIO's IO

- Describe, then interpret: HTTP endpoints using tapir
- Functional pancakes

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:

```
import sttp.tapir._

val endpoint: Endpoint[Unit, Unit, Unit, Any] = ???
```

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:

```
import sttp.tapir._

val userEndpoint: Endpoint[(UUID, Int), String, User, Any] = ???
```

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. It can also be used to access the original request through the `underlying: Any` field.

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:

```
import sttp.tapir._
import sttp.tapir.generic.auto._
import sttp.tapir.json.circe._
import io.circe.generic.auto._
import java.util.UUID
```

(continues on next page)

(continued from previous page)

```
case class User(name: String)

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], Any] =
  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:

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

case class ErrorInfo(message: String)

val baseEndpoint: Endpoint[Unit, ErrorInfo, Unit, Any] =
  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:

```
case class Status(uptime: Long)

val statusEndpoint: Endpoint[Unit, ErrorInfo, Status, Any] =
  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:

```
import sttp.tapir._
import java.util.UUID

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

val paging: EndpointInput[Paging] =
  query[UUID]("start").and(query[Option[Int]]("limit"))
    .map(input => Paging(input._1, input._2))(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[CaseClass]` method, which automatically provides the functions to construct/deconstruct the case class:

```
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 Describing input/output values using annotations

Inputs and outputs can also be built for case classes using annotations. For example, for the case class `User`

```
import sttp.tapir.EndpointIO.annotations._

case class User(
  @query
  name: String,
  @cookie
  sessionId: Long
)
```

endpoint input can be generated using macro `EndpointInput.derived[User]` which is equivalent to

```
import sttp.tapir._

val userInput: EndpointInput[User] =
  query[String]("user").and(cookie[Long]("sessionId")).mapTo[User]
```

Similarly, endpoint outputs can be derived using `EndpointOutput.derived[...]`.

Following annotations are available in package `sttp.tapir.annotations` for describing both input and output values:

- `@header` captures a header with the same name as name of annotated field in a case class. This annotation can also be used with optional parameter `@header("headerName")` in order to capture a header with name `"headerName"` if a name of header is different from name of annotated field in a case class
- `@headers` captures all headers. Can only be applied to fields represented as `List[Header]`
- `@cookies` captures all cookies. Can only be applied to fields represented as `List[Cookie]`
- `@jsonbody` captures JSON body of request or response. Can only be applied to field if there is implicit JSON `Codec` instance from `String` to target type
- `@xmlbody` captures XML body of request or response. Also requires implicit XML `Codec` instance from `String` to target type

Following annotations are only available for describing input values:

- `@query` captures a query parameter with the same name as name of annotated field in a case class. The same as annotation `@header` it has optional parameter to specify alternative name for query parameter
- `@params` captures all query parameters. Can only be applied to fields represented as `QueryParams`
- `@cookie` captures a cookie with the same name as name of annotated field in a case class. The same as annotation `@header` it has optional parameter to specify alternative name for cookie
- `@apikey` wraps any other input and designates it as an API key. Can only be used with another annotations

- `@basic` extracts data from the `Authorization` header. Can only be applied for field represented as `UsernamePassword`
- `@bearer` extracts data from the `Authorization` header removing the `Bearer` prefix.
- `@path` captures a path segment. Can only be applied to field of a case class if this case class is annotated by annotation `@endpointInput`. For example,

```
import sttp.tapir.EndpointIO.annotations._

@endpointInput("books/{year}/{genre}")
case class Book(
  @path
  genre: String,
  @path
  year: Int,
  @query
  name: String
)
```

Annotation `@endpointInput` specifies endpoint path. In order to capture a segment of the path, it must be surrounded in curly braces.

Following annotations are only available for describing output values:

- `@setCookie` sends value in header `Set-Cookie`. The same as annotation `@header` it has optional parameter to specify alternative name for cookie. Can only be applied for field represented as `CookieValueWithMeta`
- `@setCookies` sends several `Set-Cookie` headers. Can only be applied for field represented as `List[Cookie]`
- `@statusCode` sets status code for response. Can only be applied for field represented as `StatusCode`

4.5.4 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)"`.

4.5.5 Next

Read on about status codes.

4.6 Status codes

4.6.1 Arbitrary 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.

4.6.2 Fixed status code

A fixed status code can be specified using the `statusCode(code)` output.

4.6.3 Different outputs for status codes

It is also possible to specify how status codes correspond with 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:

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

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](
    oneOfMapping(StatusCode.NotFound, jsonBody[NotFound].description("not found")),
    oneOfMapping(StatusCode.Unauthorized, jsonBody[Unauthorized].description(
↪"unauthorized")),
    oneOfMapping(StatusCode.NoContent, emptyOutputAs(NoContent)),
    oneOfDefaultMapping(jsonBody[Unknown].description("unknown"))
  )
)
```

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

- 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 `oneOfMapping` and `oneOfDefaultMapping` return a value of type `OneOfMapping`. 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`.

4.6.4 One-of-mapping and type erasure

Type erasure may prevent a one-of-mapping from working properly. The following example will fail at compile time because `Right[NotFound]` and `Right[BadRequest]` will become `Right[Any]`:

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

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]](
    oneOfMapping(StatusCode.NotFound, jsonBody[Right[ServerError, NotFound]].
↪description("not found")),
    oneOfMapping(StatusCode.BadRequest, jsonBody[Right[ServerError, BadRequest]].
↪description("unauthorized")),
    oneOfMapping(StatusCode.InternalServerError, jsonBody[Left[ServerError,␣
↪UserError]].description("unauthorized")),
  )
)
// error: Constructing oneOfMapping of type scala.util.Right[repl.MdocSession.App.
↪ServerError,repl.MdocSession.App.NotFound] is not allowed because of type erasure.␣
↪Using a runtime-class-based check it isn't possible to verify that the input␣
↪matches the desired class. Please use oneOfMappingClassMatcher,␣
↪oneOfMappingValueMatcher or oneOfMappingFromMatchType instead
//     oneOfMappingValueMatcher(StatusCode.NotFound, jsonBody[Right[ServerError,␣
↪NotFound]].description("not found")) {
//                              ^
// error: Constructing oneOfMapping of type scala.util.Right[repl.MdocSession.App.
↪ServerError,repl.MdocSession.App.BadRequest] is not allowed because of type erasure.
↪ Using a runtime-class-based check it isn't possible to verify that the input␣
↪matches the desired class. Please use oneOfMappingClassMatcher,␣
↪oneOfMappingValueMatcher or oneOfMappingFromMatchType instead
//     oneOfMappingValueMatcher(StatusCode.BadRequest, jsonBody[Right[ServerError,␣
↪BadRequest]].description("unauthorized")) {
//                              ^
// error: Constructing oneOfMapping of type scala.util.Left[repl.MdocSession.App.
↪ServerError,repl.MdocSession.App.UserError] is not allowed because of type erasure.␣
↪Using a runtime-class-based check it isn't possible to verify that the input␣
↪matches the desired class. Please use oneOfMappingClassMatcher,␣
↪oneOfMappingValueMatcher or oneOfMappingFromMatchType instead
//     oneOfMappingValueMatcher(StatusCode.InternalServerError,␣
↪jsonBody[Left[ServerError, UserError]].description("unauthorized")) {
//                              ^
```

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

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

Of course, you could use `oneOfMappingValueMatcher` 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]](
    oneOfMappingFromMatchType(StatusCode.NotFound, jsonBody[Right[ServerError,␣
↪NotFound]].description("not found")),
    oneOfMappingFromMatchType(StatusCode.BadRequest, jsonBody[Right[ServerError,␣
↪BadRequest]].description("unauthorized")),
    oneOfMappingFromMatchType(StatusCode.InternalServerError,␣
↪jsonBody[Left[ServerError, UserError]].description("unauthorized"))
  )
)
```

4.6.5 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*.

4.6.6 Next

Read on about codecs.

4.7 Codecs

A `Codec[L, H, CF]` is a bi-directional mapping between low-level values of type `L` and high-level values of type `H`. Low level values are formatted as `CF`. A codec also contains the schema of the high-level value, which is used for validation and documentation.

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`. A query input requires a codec, where the low-level value is a `List[String]` (representing potentially 0, one, or multiple parameters with the given name in the URL). Hence, an implicit `Codec[List[String], Int, TextPlain]` value will be looked up when using the `query` method (which is defined in the `sttp.tapir` package).

In this example, the codec will verify that there's a single query parameter with the given name, and parse it as an integer. 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).

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

4.7.2 Schemas

A codec contains a schema, which describes the high-level type. The schema is used when generating documentation and enforcing validation rules.

Schema consists of:

- the schema type, which is one of the values defined in `SchemaType`, such as `SString`, `SBinary`, `SArray` or `SProduct`/`SCoproduct` (for ADTs). This is the shape of the encoded value - as it is sent over the network
- meta-data: value optionality, description, example, default value and low-level format name
- validation rules

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.

Schemas are typically referenced indirectly through codecs, and are specified when the codec is created.

As part of deriving a codec, to support a custom or complex type (e.g. for json mapping), schemas can be looked up implicitly and derived as well. See custom types for more details.

4.7.3 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 type is allowed. For example, the input/output described by `jsonBody[T]` requires a json codec.

4.7.4 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 Creating an implicit codec by hand

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

- `encode` and `rawDecode` methods
- schema (for documentation and validation)
- 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:

```
import scala.util._

class MyId private (id: String) {
  override def toString(): String = id
}
object MyId {
  def parse(id: String): Try[MyId] = {
    Success(new MyId(id))
  }
}
```

```
import sttp.tapir._
import sttp.tapir.CodecFormat.TextPlain

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:

```
import sttp.tapir.Codec.PlainCodec

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]` implicits (see the next section)

Note the derivation of e.g. circe json encoders/decoders and tapir schemas are separate processes, and must be configured separately.

4.8.3 Next

Read on about deriving schemas.

4.9 Schema derivation

Implicit schemas for basic types (`String`, `Int`, etc.), and their collections (`Option`, `List`, `Array` etc.) are defined out-of-the box. They don't contain any meta-data, such as descriptions or example values.

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

There are two policies of custom type derivation are available:

- automatic derivation
- semi automatic derivation

4.9.1 Automatic derivation

Case classes, traits and their children are recursively derived by Magnolia.

Importing `sttp.tapir.generic.auto._` (or extending the `SchemaDerivation` trait) enables fully automatic derivation for `Schema`:

```
import sttp.tapir.Schema
import sttp.tapir.generic.auto._

case class Parent(child: Child)
case class Child(value: String)

// implicit schema used by codecs
implicitly[Schema[Parent]]
```

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 implicit schemas for them. Using these, the rest will be derived automatically.

Note that when using datatypes integrations, respective codecs must also be imported to enable the derivation, e.g. for newtype you'll have to add `import sttp.tapir.codec.newtype._` or extend `TapirCodecNewType`.

4.9.2 Semi-automatic derivation

Semi-automatic derivation can be done using `Schema.derived[T]`.

It only derives selected type `T`. However, derivation is not recursive: schemas must be explicitly defined for every child type.

This mode is easier to debug and helps to avoid issues encountered by automatic mode (wrong schemas for value classes or custom types):

```
import sttp.tapir.Schema

case class Parent(child: Child)
case class Child(value: String)

implicit lazy val sChild: Schema[Child] = Schema.derived
implicit lazy val sParent: Schema[Parent] = Schema.derived
```

Note that while schemas for regular types can be safely defined as `val`s, in case of recursive values, the schema values must be `lazy val`s.

4.9.3 Derivation for recursive types in Scala3

In Scala3, any schemas for recursive types need to be provided as typed `implicit def` (not a `given`)! For example:

```
case class RecursiveTest(data: List[RecursiveTest])
object RecursiveTest {
  implicit def f1Schema: Schema[RecursiveTest] = Schema.derived[RecursiveTest]
}
```

The implicit doesn't have to be defined in the companion object, just anywhere in scope. This applies to cases where the schema is looked up implicitly, e.g. for `jsonBody`.

4.9.4 Configuring derivation

It is possible to configure Magnolia's automatic derivation to use `snake_case`, `kebab-case` or a custom field naming policy, by providing an implicit `sttp.tapir.generic.Configuration` value. This influences how the low-level representation is described in documentation:

```
import sttp.tapir.generic.Configuration

implicit val customConfiguration: Configuration =
  Configuration.default.withSnakeCaseMemberNames
```

4.9.5 Manually providing schemas

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`:

```
import sttp.tapir._

case class MyCustomType()
implicit val schemaForMyCustomType: Schema[MyCustomType] = Schema.string
// or, if the low-level representation is e.g. a number
implicit val anotherSchemaForMyCustomType: Schema[MyCustomType] = Schema(SchemaType.
↪SInteger())
```

4.9.6 Sealed traits / coproducts

Schema derivation for coproduct types (sealed trait hierarchies) is supported as well. By default, such hierarchies will be represented as a coproduct which contains a list of child schemas, without any discriminator field.

A discriminator field can be specified for coproducts by providing it in the configuration; this will be only used during automatic and semi-automatic derivation:

```
import sttp.tapir.generic.Configuration

implicit val customConfiguration: Configuration =
  Configuration.default.withDiscriminator("who_am_i")
```

Alternatively, derived schemas can be customised (see below), and a discriminator can be added by calling the `SchemaType.SCoproduct.addDiscriminatorField(name, schema, maping)` method.

Finally, if the discriminator is a field that's defined on the base trait (and hence in each implementation), the schemas can be specified using `Schema.oneOfUsingField`, for example (this will also generate the appropriate mappings):

```
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"
}

import sttp.tapir._

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

4.9.7 Customising derived schemas

Using annotations

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. One way the automatic & semi-automatic derivation can be customised is using annotations:

- `@encodedName` sets name for case class's field which is used in the encoded form (and also in documentation)
- `@description` sets description for the whole case class or its field
- `@default` sets default value for a case class field
- `@encodedExample` sets example value for a case class field which is used in the documentation in the encoded form
- `@format` sets the format for a case class field
- `@deprecated` marks a case class's field as deprecated

These annotations will adjust schemas, after they are looked up using the normal implicit mechanisms.

Using implicits

If the target type isn't accessible or can't be modified, schemas can be customized by looking up an implicit instance of the `Derived[Schema[T]]` type, modifying the value, 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:

```
import sttp.tapir._
import sttp.tapir.generic.auto._
import sttp.tapir.generic.Derived

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`.

Using value classes/tagged types

An alternative to customising schemas for case class fields of primitive type (e.g. `Int`s), is creating a unique type. As schema lookup is type-driven, if a schema for a such type is provided as an implicit value, it will be used during automatic or semi-automatic schema derivation. Such schemas can have custom meta-data, including description, validation, etc.

To introduce unique types for primitive values, which don't have a runtime overhead, 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 with validator:

```
import sttp.tapir._
import sttp.tapir.generic.auto._
import sttp.tapir.json.circe._
import io.circe.{ Encoder, Decoder }
import io.circe.generic.semiauto._

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

implicit val amountSchema: Schema[Amount] = Schema(SchemaType.SInteger()).
↪validate(Validator.min(1).contramap(_.v))
implicit val amountEncoder: Encoder[Amount] = Encoder.encodeInt.contramap(_.v)
implicit val amountDecoder: Decoder[Amount] = Decoder.decodeInt.map(Amount.apply)

implicit val decoder: Decoder[FruitAmount] = deriveDecoder[FruitAmount]
implicit val encoder: Encoder[FruitAmount] = deriveEncoder[FruitAmount]

val e: Endpoint[FruitAmount, Unit, Unit, Nothing] =
  endpoint.in(jsonBody[FruitAmount])
```

4.9.8 Next

Read on about validation.

4.10 Validation

Tapir supports validation for primitive 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.

4.10.1 Single type validation

Validation rules are part of the `Schema` for a given type, and can be added either directly to the schema, or via the `Codec` or `EndpointInput`/`EndpointOutput`. For example, when defining a codec for a type, we have the `.validate()` method:

```
import sttp.tapir._
import sttp.tapir.CodecFormat.TextPlain

case class MyId(id: String)

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

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

```
import sttp.tapir._

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

For optional/iterable inputs/outputs, to validate the contained value(s), use:

```
import sttp.tapir._

query[Option[Int]]("item").validateOption(Validator.min(0))
query[List[Int]]("item").validateIterable(Validator.min(0)) // validates each␣
↪repeated parameter
```

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

4.10.2 Validation rules and automatic codec derivation

As validators are parts of schemas, they are looked up as part of the implicit `Schema[T]` values. When customising schemas, use the `.validate` method on the schema to add a validator.

4.10.3 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*.

4.10.4 Enum validators

Validators for enumerations can be created using:

- `Validator.derivedEnumeration`, which takes a type parameter. This should be an abstract, sealed base type, and using a macro determines the possible values
- `Validator.enumeration`, which 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 can be specified by:

- explicitly providing it using the overloaded `enumeration` method with an `encode` parameter
- by using one of the `.encode` methods on the `Validator.Enumeration` instance
- when the values possible values are of a basic type (numbers, strings), the encode function is inferred if not present
- by adding the validator directly to a codec using `.validate` (the encode function is then taken from the codec)

To simplify creation of schemas and codec, with a derived enum validator, `Schema.derivedEnumeration` and `Codec.derivedEnumeration` helper methods are available. For example:

```
import sttp.tapir._
import sttp.tapir.Codec.PlainCodec

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

implicit def plainCodecForColor: PlainCodec[Color] = {
  Codec.derivedEnumeration[String, Color](
    (_: String) match {
      case "red"  => Some(Red)
      case "blue" => Some(Blue)
      case _      => None
    },
    _.toString.toLowerCase
  )
}
```

If the enum is nested within an object and its values aren't of a "basic" type (numbers, strings), regardless of whether the codec for that object is defined by hand or derived, we need to specify the encode function by hand:

```
// providing the enum values by hand
implicit def colorSchema: Schema[Color] = Schema.string.validate(
  Validator.enumeration(List(Blue, Red), (c: Color) => Some(c.toString.toLowerCase)))

// or deriving the enum values and using the helper function
implicit def colorSchema2: Schema[Color] = Schema.derivedEnumeration[Color](encode =␣
↪Some(_.toString.toLowerCase))
```

4.10.5 Next

Read on about content types.

4.11 Content type

The endpoint's output content type is bound to the body outputs, that are specified for the endpoint (if any). The codec of a body output contains a `CodecFormat`, which in turns contains the `MediaType` instance.

4.11.1 Codec formats and server interpreters

Codec formats define the *default* media type, which will be set as the `Content-Type` header. However, any user-provided value will override this default:

- dynamic content type, using `.out(header(HeaderNames.ContentType))`
- fixed content type, using e.g. `out(header(Header.contentType(MediaType.ApplicationJson)))`

4.11.2 Multiple content types

Multiple, alternative content types can be specified using `oneOf`, similarly as when specifying status code mappings.

On the server side, the appropriate mapping will be chosen using content negotiation, via the `Accept` header, using the *configurable* `ContentTypeInterceptor`. Note that both the base media type, and the charset for text types are taken into account.

On the client side, the appropriate mapping will be chosen basing on the `Content-Type` header value.

For example:

```
import sttp.tapir._
import sttp.tapir.Codec.{JsonCodec, XmlCodec}
import sttp.model.StatusCode

case class Entity(name: String)
implicit val jsonCodecForOrganization: JsonCodec[Entity] = ???
implicit val xmlCodecForOrganization: XmlCodec[Entity] = ???

endpoint.out(
  oneOf(
    oneOfMapping(StatusCode.Ok, customJsonBody[Entity]),
    oneOfMapping(StatusCode.Ok, xmlBody[Entity])
  )
)
```

For details on how to create codes manually or derive them automatically, see custom types and the subsequent section on json.

4.11.3 Next

Read on about json support.

4.12 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, Play JSON,

Tethys JSON, Jsoniter-scala, and Json4s 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 `customJsonBody[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.

4.12.1 Circe

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

```
"com.softwaremill.sttp.tapir" %% "tapir-json-circe" % "0.19.0-M9"
```

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 sttp.tapir.generic.auto._
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:

```
import io.circe._

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`:

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

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"}
```

4.12.2 µPickle

To use µPickle add the following dependency to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-json-upickle" % "0.19.0-M9"
```

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 sttp.tapir.generic.auto._
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

4.12.3 Play JSON

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

```
"com.softwaremill.sttp.tapir" %% "tapir-json-play" % "0.19.0-M9"
```

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.

4.12.4 Spray JSON

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

```
"com.softwaremill.sttp.tapir" %% "tapir-json-spray" % "0.19.0-M9"
```

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.

4.12.5 Tethys JSON

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

```
"com.softwaremill.sttp.tapir" %% "tapir-json-tethys" % "0.19.0-M9"
```

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.

4.12.6 Jsoniter Scala

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

```
"com.softwaremill.sttp.tapir" %% "tapir-jsoniter-scala" % "0.19.0-M9"
```

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.

4.12.7 Json4s

To use json4s add the following dependencies to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-json-json4s" % "0.19.0-M9"
```

And one of the implementations:

```
"org.json4s" %% "json4s-native" % "4.0.3"
// Or
"org.json4s" %% "json4s-jackson" % "4.0.3"
```

Next, import the package (or extend the `TapirJson4s` trait, see *MyTapir* and add `TapirJson4s` instead of `TapirCirceJson`):

```
import sttp.tapir.json.json4s._
```

Json4s requires `Serialization` and `Formats` implicit values in scope, for example:

```
import org.json4s._
// ...
implicit val serialization: Serialization = org.json4s.jackson.Serialization
implicit val formats: Formats = org.json4s.jackson.Serialization.formats(NoTypeHints)
```

4.12.8 Zio JSON

To use Zio JSON, add the following dependency to your project:

```
"com.softwaremill.sttp.tapir" %% "tapir-json-zio" % "0.19.0-M9"
```

Next, import the package (or extend the `TapirJsonZio` trait, see *MyTapir* and add `TapirJsonZio` instead of `TapirCirceJson`):

```
import sttp.tapir.json.zio._
```

Zio JSON requires `JsonEncoder` and `JsonDecoder` implicit values in scope for each type you want to serialize.

4.12.9 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).

4.12.10 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 `Schema[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.

4.12.11 Next

Read on about working with forms.

4.13 Forms

4.13.1 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):

```
import sttp.tapir._

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

```
import sttp.tapir._
import sttp.tapir.generic.auto._

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

formBody[RegistrationForm]: EndpointIO.Body[String, 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`, or customised using the `@encodedName` annotation.

4.13.2 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:

```
import sttp.tapir._
import sttp.model.Part

multipartBody: EndpointIO.Body[Seq[RawPart], Seq[Part[Array[Byte]]]]
```

`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]`). Custom multipart codecs can be defined with the `Codec.multipartCodec` method, and then used with `multipartBody[T]`.

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`, or customised using the `@encodedName` annotation.

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:

```
import sttp.tapir._
import sttp.model.Part
import java.io.File
```

(continues on next page)

(continued from previous page)

```
import sttp.tapir.generic.auto._

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

multipartBody[RegistrationForm]: EndpointIO.Body[Seq[RawPart], RegistrationForm]
```

4.13.3 Next

Read on about authentication.

4.14 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.

For each `auth` scheme, one can define `WWW-Authenticate` headers that should be returned by the server in case input is not provided. Default behavior is to return `401` status with the headers needed for authentication.

For example if you define `endpoint.get.in("path").in(auth.basic[UsernamePassword]())` then your browser will show you a password prompt.

4.14.1 Next

Read on about streaming support.

4.15 Streaming support

Both input and output bodies can be mapped to a stream, by using `stream[*]Body(streams)`. The parameter `streams` must implement the `Streams[S]` capability, and determines the precise type of the binary stream supported by the given non-blocking streams implementation. The interpreter must then support the given capability. Refer to the documentation of server/client interpreters for more information.

Note: Here, streams refer to asynchronous, non-blocking, "reactive" stream implementations, such as akka-streams, fs2 or zio-streams. If you'd like to use blocking streams (such as `InputStream`), these are available through e.g. `inputStreamBody` without any additional requirements on the interpreter.

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, a schema must be provided for documentation. By default, when using `streamBinaryBody`, the schema will simply be that of a binary body. If you have a textual stream, you can use `streamTextBody`. In that case, you'll also need to provide the default format (media type) and optional charset to be used to determine the content type.

To provide an arbitrary schema, use `streamBody`. Note, however, that this schema will only be used for generating documentation. The incoming stream data will not be validated using the schema validators.

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:

```
import sttp.tapir._
import sttp.tapir.generic.auto._
import sttp.capabilities.akka.AkkaStreams
import akka.stream.scaladsl._
import akka.util.ByteString

case class Person(name: String)

// copying the derived json schema type
endpoint.out(streamBody(AkkaStreams)(Schema.derived[List[Person]], CodecFormat.
↪Json()))
```

See also the *runnable streaming example*.

4.15.1 Next

Read on about web sockets.

4.16 Web sockets

Web sockets are supported through stream pipes, converting a stream of incoming messages to a stream of outgoing messages. That's why web socket endpoints require both the `Streams` and `WebSocket` capabilities (see streaming support for more information on streams).

4.16.1 Typed web sockets

Web sockets outputs can be used in two variants. In the first, both requests and responses are handled by codecs. Typically, a codec handles either text or binary messages, signalling decode failure (which closes the web socket), if an unsupported frame is passed to decoding.

For example, here's an endpoint where the requests are strings (hence only text frames are handled), and the responses are parsed/formatted as json:

```
import sttp.tapir._
import sttp.capabilities.akka.AkkaStreams
import sttp.tapir.json.circe._
import sttp.tapir.generic.auto._
import io.circe.generic.auto._

case class Response(msg: String, count: Int)
endpoint.out(
  webSocketBody[String, CodecFormat.TextPlain, Response, CodecFormat.
↪Json](AkkaStreams))
```

When creating a `webSocketBody`, we need to provide the following parameters:

- the type or requests, along with its codec format (which is used to lookup the appropriate codec, as well as determines the media type in documentation)
- the type of responses, along with its codec format
- the `Streams` implementation, which determines the pipe type

By default, ping-pong frames are handled automatically, fragmented frames are combined, and close frames aren't decoded, but this can be customised through methods on `webSocketBody`.

4.16.2 Raw web sockets

Alternatively, it's possible to obtain a raw pipe transforming `WebSocketFrames`:

```
import akka.stream.scaladsl.Flow
import sttp.tapir._
import sttp.capabilities.akka.AkkaStreams
import sttp.capabilities.WebSockets
import sttp.ws.WebSocketFrame

endpoint.out(webSocketBodyRaw(AkkaStreams)): Endpoint[
  Unit,
  Unit,
  Flow[WebSocketFrame, WebSocketFrame, Any],
  AkkaStreams with WebSockets]
```

Such a pipe by default doesn't handle ping-pong frames automatically, doesn't concatenate fragmented flames, and passes close frames to the pipe as well. As before, this can be customised by methods on the returned output.

Request/response schemas can be customised through `.requestsSchema` and `.responsesSchema`.

4.16.3 Interpreting as a sever

When interpreting a web socket endpoint as a server, the *server logic* needs to provide a streaming-specific pipe from requests to responses. E.g. in akka's case, this will be `Flow[REQ, RESP, Any]`.

Refer to the documentation of interpreters for more details, as not all interpreters support all settings.

4.16.4 Interpreting as a client

When interpreting a web socket endpoint as a client, after applying the input parameters, the result is a pipe representing message processing as it happens on the server.

Refer to the documentation of interpreters for more details, as there are interpreter-specific additional requirements.

4.16.5 Interpreting as documentation

Web socket endpoints can be interpreted into *AsyncAPI documentation*.

4.16.6 Determining if the request is a web socket upgrade

The `isWebSocket` endpoint input can be used to determine if the request contains the web socket upgrade headers. The input only impacts server interpreters, doesn't affect documentation and its value is discarded by client interpreters.

4.16.7 Next

Read on about datatypes integrations.

4.17 Datatypes integrations

4.17.1 Cats datatypes integration

The `tapir-cats` module contains additional instances for some cats datatypes as well as additional syntax:

```
"com.softwaremill.sttp.tapir" %% "tapir-cats" % "0.19.0-M9"
```

- `import sttp.tapir.integ.cats.codec._` - brings schema, validator and codec instances
- `import sttp.tapir.integ.cats.syntax._` - brings additional syntax for `tapir` types

4.17.2 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" % "0.19.0-M9"
```

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!).

4.17.3 Enumeratum integration

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

```
"com.softwaremill.sttp.tapir" %% "tapir-enumeratum" % "0.19.0-M9"
```

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`.

4.17.4 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.

```
import sttp.tapir._

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)

    // needs to be a def or lazy val so that the enumeration values are available!
    implicit def schemaForEnum: Schema[e.Value] = Schema.string.validate(Validator.
↪enumeration(e.values.toList, v => Option(v)))
}
object Color extends Enumeration with EnumHelper {
    type Color = Value
    val Blue = Value("blue")
    val Red   = Value("red")
}
```

4.17.5 NewType integration

If you use scala-newtype, the `tapir-newtype` module will provide implicit codecs and schemas for a types with a `@newtype` and `@newsubtype` annotations as long as a codec and schema for its underlying value already exists:

```
"com.softwaremill.sttp.tapir" %% "tapir-newtype" % "0.19.0-M9"
```

Then, `import sttp.tapir.codec.newtype._`, or extend the `sttp.tapir.codec.enumeratum.TapirCodecNewType` trait to bring the implicit values into scope.

4.17.6 Derevo integration

The `tapir-derevo` module provides a way to derive schema for your type using `@derive` annotation. For details refer to derevo documentation. To use, add the following dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-derevo" % "0.19.0-M9"
```

Then you can derive schema for your ADT along with other typeclasses besides ADT declaration itself:

```
import derevo.derive
import sttp.tapir.derevo.schema

@derive(schema)
case class Person(name: String, age: Int)

//or with custom description

@derive(schema("Type of currency in the country"))
sealed trait Currency
object Currency {
    case object CommunisticCurrency extends Currency
    case class USD(amount: Long) extends Currency
}
```

The annotation will simply generate a `Schema[T]` for your type `T` and put it into companion object. Generation rules are the same as in `Schema.derived[T]`.

This will also work for newtypes — estatico or supertagged:

```
import derevo.derive
import sttp.tapir.derevo.schema
import io.estatico.newtype.macros.newtype

object types {

  @derive(schema)
  @newtype
  case class Amount(i: Int)
}
```

Resulting schema will be equivalent to `implicitly[Schema[Int]].map(i => Some(types.Amount(i)))`. Note that due to limitations of the `derevo` library one can't provide custom description for generated schema.

4.17.7 Next

Read on about serving static content.

4.18 Serving static content

Tapir contains predefined endpoints, server logic and server endpoints which allow serving static content, originating from local files or application resources. These endpoints respect etags as well as if-modified-since headers.

4.18.1 Files

The easiest way to expose static content from the local filesystem is to use the `filesServerEndpoint`. This method is parametrised with the path, at which the content should be exposed, as well as the local system path, from which to read the data.

Such an endpoint has to be interpreted using your server interpreter. For example, using the akka-http interpreter:

```
import akka.http.scaladsl.server.Route

import sttp.tapir._
import sttp.tapir.server.akkahttp.AkkaHttpServerInterpreter

import scala.concurrent.Future

val filesRoute: Route = AkkaHttpServerInterpreter().toRoute(
  filesServerEndpoint[Future]("site" / "static")("/home/static/data")
)
```

Using the above endpoint, a request to `/site/static/css/styles.css` will try to read the `/home/static/data/css/styles.css` file.

A single file can be exposed using `fileServerEndpoint`.

4.18.2 Resources

Similarly, the `resourcesServerEndpoint` can be used to expose the application's resources at the given prefix.

A single resource can be exposed using `resourceServerEndpoint`.

4.18.3 Endpoint description and server logic

The descriptions of endpoints which should serve static data, and the server logic which implements the actual file/resource reading are also available separately for further customisation.

The `filesEndpoint` and `resourcesEndpoint` are descriptions which contain the metadata (including caching headers) required to serve a file or resource, and possible error outcomes. This is captured using the `StaticInput`, `StaticErrorOuput` and `StaticOutput[T]` classes.

The `sttp.tapir.static.Files` and `sttp.tapir.static.Resources` objects contain the logic implementing server-side reading of files or resources, with etag/last modification support.

4.19 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" % "0.19.0-M9"
```

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" % "0.19.0-M9" exclude("com.
↪typesafe.akka", "akka-stream_2.12")
```

Now import the object:

```
import sttp.tapir.server.akkahttp.AkkaHttpServerInterpreter
```

The `AkkaHttpServerInterpreter` objects contains methods such as: `toRoute`, `toRouteRecoverErrors` and `toDirective`.

4.19.1 Using `toRoute` and `toRouteRecoverErrors`

The `toRoute` method requires the logic of the endpoint to be given as a function of type:

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

The `toRouteRecoverErrors` method 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.AkkaHttpServerInterpreter
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, Any] =
  endpoint.in(stringBody).out(plainBody[Int])

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

Note that the second argument to `toRoute` is a function with one argument, 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`:

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

def logic(s: String, i: Int): Future[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Any] = ???
val aRoute: Route = AkkaHttpServerInterpreter().toRoute(anEndpoint)((logic _).tupled)
```

4.19.2 Combining directives

The tapir-generated `Route` captures from the request only what is described by the endpoint. Combine with other akka-http directives to add additional behavior, or get more information from the request.

For example, wrap the tapir-generated route in a metrics route, or nest a security directive in the tapir-generated directive.

Edge-case endpoints, which require special logic not expressible using tapir, can be implemented directly using akka-http. For example:

```
import sttp.tapir._
import sttp.tapir.server.akkahttp.AkkaHttpServerInterpreter
import akka.http.scaladsl.server._

case class User(email: String)
def metricsDirective: Directive0 = ???
def securityDirective: Directive1[User] = ???
val tapirEndpoint: Endpoint[String, Unit, Unit, Any] = endpoint.in(path[String]("input
↪"))
```

(continues on next page)

(continued from previous page)

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

4.19.3 Streaming

The akka-http interpreter accepts streaming bodies of type `Source[ByteString, Any]`, as described by the `AkkaStreams` capability. Both response bodies and request bodies can be streamed. Usage: `streamBody(AkkaStreams)(schema, format)`.

The capability can be added to the classpath independently of the interpreter through the `"com.softwaremill.sttp.shared" %% "akka"` dependency.

4.19.4 Web sockets

The interpreter supports web sockets, with pipes of type `Flow[REQ, RESP, Any]`. See *web sockets* for more details.

akka-http does not expose control frames (`Ping`, `Pong` and `Close`), so any setting regarding them are discarded, and ping/pong frames which are sent explicitly are ignored. Automatic pings can be instead enabled through configuration.

4.19.5 Server Sent Events

The interpreter supports SSE (Server Sent Events).

For example, to define an endpoint that returns event stream:

```
import akka.stream.scaladsl.Source
import sttp.model.sse.ServerSentEvent
import sttp.tapir._
import sttp.tapir.server.akkahttp.{AkkaHttpServerInterpreter, serverSentEventsBody}

import scala.concurrent.Future

val sseEndpoint = endpoint.get.out(serverSentEventsBody)

val routes = AkkaHttpServerInterpreter().toRoute(sseEndpoint)(_ =>
  Future.successful(Right(Source.single(ServerSentEvent(Some("data"), None, None,␣
↪None))))
)
```

4.19.6 Configuration

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

4.19.7 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 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" % "0.19.0-M9"
```

and import the object:

```
import sttp.tapir.server.http4s.Http4sServerInterpreter
```

This objects contains the `toRoutes` and `toRoutesRecoverErrors` methods. 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.Http4sServerInterpreter
import cats.effect.IO
import org.http4s.HttpRoutes

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

val countCharactersEndpoint: Endpoint[String, Unit, Int, Any] =
  endpoint.in(stringBody).out(plainBody[Int])
val countCharactersRoutes: HttpRoutes[IO] =
  Http4sServerInterpreter[IO]().toRoutes(countCharactersEndpoint)(countCharacters _)
```

Note that the second argument to `toRoute` is a function with one argument, 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`:

```
import sttp.tapir._
import sttp.tapir.server.http4s.Http4sServerInterpreter
import cats.effect.IO
import org.http4s.HttpRoutes

def logic(s: String, i: Int): IO[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Any] = ???
val routes: HttpRoutes[IO] = Http4sServerInterpreter[IO]().
↪toRoutes(anEndpoint)((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.

4.20.1 Streaming

The http4s interpreter accepts streaming bodies of type `Stream[F, Byte]`, as described by the `Fs2Streams` capability. Both response bodies and request bodies can be streamed. Usage: `streamBody(Fs2Streams[F])(schema, format)`.

The capability can be added to the classpath independently of the interpreter through the `"com.softwaremill.sttp.shared" %% "http4s"` dependency.

4.20.2 Web sockets

The interpreter supports web sockets, with pipes of type `Pipe[F, REQ, RESP]`. See *web sockets* for more details.

4.20.3 Server Sent Events

The interpreter supports SSE (Server Sent Events).

For example, to define an endpoint that returns event stream:

```
import cats.effect.IO
import sttp.model.sse.ServerSentEvent
import sttp.tapir._
import sttp.tapir.server.http4s.{Http4sServerInterpreter, serverSentEventsBody}

val sseEndpoint = endpoint.get.out(serverSentEventsBody[IO])

val routes = Http4sServerInterpreter[IO]().toRoutes(sseEndpoint)(_ =>
  IO(Right(fs2.Stream(ServerSentEvent(Some("data"), None, None, None))))
)
```

4.20.4 Configuration

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

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

4.20.5 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.

4.21 Running as an http4s server using ZIO

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 an interpreter useful when exposing the endpoints using the http4s server.

You'll need the following dependency for the `ZEndpoint`, `ZServerEndpoint` type aliases and helper classes:

```
"com.softwaremill.sttp.tapir" %% "tapir-zio" % "0.19.0-M9"
```

or just add the zio-http4s integration which already depends on `tapir-zio`:

```
"com.softwaremill.sttp.tapir" %% "tapir-zio-http4s-server" % "0.19.0-M9"
```

Next, instead of the usual `import sttp.tapir._`, you should import (or extend the `ZTapir` trait, see *MyTapir*):

```
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._`.

4.21.1 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.

4.21.2 Exposing endpoints using the http4s server

To interpret a `ZServerEndpoint` as a http4s server, use the following interpreter:

```
import sttp.tapir.server.http4s.ztapir.ZHttp4sServerInterpreter
```

To help with type-inference, you first need to call `ZHttp4sServerInterpreter().from()` providing:

- an endpoint and logic: `def from[I, E, O](e: ZEndpoint[I, E, O])(logic: I => ZIO[R, E, O])`
- a single server endpoint: `def from[I, E, O](se: ZServerEndpoint[R, I, E, O])`
- multiple server endpoints: `def from(serverEndpoints: List[ZServerEndpoint[R, _, _, _]])`

Then, call `.toRoutes` to obtain the http4s `HttpRoutes` instance.

Note that the resulting `HttpRoutes` always require `Clock` and `Blocking` in the environment.

If you have multiple endpoints with different environmental requirements, the environment must be first widened so that it is uniform across all endpoints, using the `.widen` method:

```
import org.http4s.HttpRoutes
import sttp.tapir.ztapir._
import sttp.tapir.server.http4s.ztapir.ZHttp4sServerInterpreter
import zio.{Has, RIO, ZIO}
import zio.blocking.Blocking
import zio.clock.Clock
import zio.interop.catz._
```

(continues on next page)

(continued from previous page)

```
trait Component1
trait Component2
type Service1 = Has[Component1]
type Service2 = Has[Component2]

val serverEndpoint1: ZServerEndpoint[Service1, Unit, Unit, Unit] = ???
val serverEndpoint2: ZServerEndpoint[Service2, Unit, Unit, Unit] = ???

type Env = Service1 with Service2
val routes: HttpRoutes[RIO[Env with Clock with Blocking, *]] =
  ZHttp4sServerInterpreter().from(List(
    serverEndpoint1.widen[Env],
    serverEndpoint2.widen[Env]
  )).toRoutes // this is where zio-cats interop is needed
```

4.21.3 Streaming

The http4s interpreter accepts streaming bodies of type `zio.stream.Stream[Throwable, Byte]`, as described by the `ZioStreams` capability. Both response bodies and request bodies can be streamed. Usage: `streamBody(ZioStreams)(schema, format)`.

The capability can be added to the classpath independently of the interpreter through the `"com.softwaremill.sttp.shared" %% "zio"` or `tapir-zio` dependency.

4.21.4 Web sockets

The interpreter supports web sockets, with pipes of type `zio.stream.Stream[Throwable, REQ] => zio.stream.Stream[Throwable, RESP]`. See *web sockets* for more details.

4.21.5 Server Sent Events

The interpreter supports SSE (Server Sent Events).

For example, to define an endpoint that returns event stream:

```
import sttp.model.sse.ServerSentEvent
import sttp.tapir.server.http4s.ztapir.{ZHttp4sServerInterpreter,␣
↪serverSentEventsBody}
import sttp.tapir.ztapir._
import org.http4s.HttpRoutes
import zio.{UIO, RIO}
import zio.blocking.Blocking
import zio.clock.Clock
import zio.stream.Stream

val sseEndpoint: ZEndpoint[Unit, Unit, Stream[Throwable, ServerSentEvent]] = endpoint.
↪get.out(serverSentEventsBody)

val routes: HttpRoutes[RIO[Clock with Blocking, *]] = ZHttp4sServerInterpreter()
  .from(sseEndpoint.zServerLogic(_ => UIO(Stream(ServerSentEvent(Some("data"), None,␣
↪None, None)))))
  .toRoutes
```

4.21.6 Examples

Three examples of using the ZIO integration are available. The first two showcase basic functionality, while the third shows how to use partial server logic methods:

- ZIO basic example
- ZIO environment example
- ZIO partial server logic example

4.22 Running as a Netty-based server

Warning! This is an early development module, with important features missing.

To expose an endpoint using a Netty-based server, first add the following dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-netty-server" % "0.19.0-M9"
```

Then, use `NettyServer().addEndpoints` to expose `Future`-based server endpoints.

For example:

```
import sttp.tapir._
import sttp.tapir.server.netty.{NettyServer, NettyServerBinding}

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

val helloWorld = endpoint
  .get
  .in("hello").in(query[String]("name"))
  .out(stringBody)
  .serverLogic(name => Future.successful[Either[Unit, String]](Right(s"Hello, $name!
↪")))

val binding: Future[NettyServerBinding] = NettyServer().addEndpoint(helloWorld).
↪start()
```

4.22.1 Configuration

The interpreter can be configured by providing an `NettyServerOptions` value, see server options for details.

Some of the options can be configured directly using a `NettyServer` instance, such as the host and port. Others can be passed using the `NettyServer(options)` methods. Options may also be overriden when adding endpoints.

4.22.2 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.

4.23 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" % "0.19.0-M9"
```

and import the object:

```
import sttp.tapir.server.finatra.FinatraServerInterpreter
```

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

```
"com.softwaremill.sttp.tapir" %% "tapir-finatra-server-cats" % "0.19.0-M9"
```

and import the object:

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

The interpreter objects contain the `toRoute` and `toRouteRecoverErrors` method. 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 giben 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.{ FinatraServerInterpreter, FinatraRoute }
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, Any] =
  endpoint.in(stringBody).out(plainBody[Int])

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

or a cats-effect's example:

```
import cats.effect.IO
import cats.effect.std.Dispatcher
import sttp.tapir._
import sttp.tapir.server.finatra.FinatraRoute
import sttp.tapir.server.finatra.cats.FinatraCatsServerInterpreter

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

(continues on next page)

(continued from previous page)

```
val countCharactersEndpoint: Endpoint[String, Unit, Int, Any] =
  endpoint.in(stringBody).out(plainBody[Int])

def dispatcher: Dispatcher[IO] = ???

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

Note that the second argument to `toRoute` is a function with one argument, 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`:

```
import sttp.tapir._
import sttp.tapir.server.finatra.{ FinatraServerInterpreter, FinatraRoute }
import com.twitter.util.Future

def logic(s: String, i: Int): Future[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Any] = ???
val aRoute: FinatraRoute = FinatraServerInterpreter().toRoute(anEndpoint)((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`.

```
import sttp.tapir.server.finatra._
import com.twitter.finatra.http.Controller

val aRoute: FinatraRoute = ???
class MyController extends Controller with TapirController {
  addTapirRoute(aRoute)
}
```

4.23.1 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.

4.24 Running as a Play server

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

```
"com.softwaremill.sttp.tapir" %% "tapir-play-server" % "0.19.0-M9"
```

and (if you don't already depend on Play)

```
"com.typesafe.play" %% "play-akka-http-server" % "2.8.7"
```

or

```
"com.typesafe.play" %% "play-netty-server" % "2.8.7"
```

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

Then import the object:

```
import sttp.tapir.server.play.PlayServerInterpreter
```

This object contains the `toRoute` and `toRoutesRecoverError` methods. 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.PlayServerInterpreter
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
import akka.stream.Materializer
import play.api.routing.Router.Routes

implicit val materializer: Materializer = ???

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

val countCharactersEndpoint: Endpoint[String, Unit, Int, Any] =
  endpoint.in(stringBody).out(plainBody[Int])
val countCharactersRoutes: Routes =
  PlayServerInterpreter().toRoutes(countCharactersEndpoint)(countCharacters _)
```

Note that the second argument to `toRoutes` is a function with one argument, 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`:

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

implicit val materializer: Materializer = ???

def logic(s: String, i: Int): Future[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Any] = ???
val aRoute: Routes = PlayServerInterpreter().toRoutes(anEndpoint)((logic _).tupled)
```

4.24.1 Bind the routes

Creating the HTTP server manually

An HTTP server can then be started as in the following example:

```
import play.core.server._
import play.api.routing.Router.Routes

val aRoute: Routes = ???

object Main {
```

(continues on next page)

(continued from previous page)

```
  // 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 =>
      aRoute
    }
  }
}
```

As part of an existing Play application

Or, if you already have an existing Play application, you can create a `Router` class and bind it to the application.

First, add a line like following in the `routes` files:

```
->      /api                        api.ApiRouter
```

Then create a class like this:

```
class ApiRouter @Inject() () extends SimpleRouter {
  override def routes: Routes = {
    anotherRoutes.orElse(tapirGeneratedRoutes)
  }
}
```

Find more details about how to bind a `Router` to your application in the Play framework documentation.

4.24.2 Configuration

The interpreter can be configured by providing a `PlayServerOptions` value, see server options for details.

4.24.3 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.

4.25 Running as a Vert.X server

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

Vert.x interpreter can be used with different effect systems (cats-effect, ZIO) as well as Scala's standard `Future`.

4.25.1 Scala's standard `Future`

Add the following dependency

```
"com.softwaremill.sttp.tapir" %% "tapir-vertx-server" % "0.19.0-M9"
```

to use this interpreter with `Future`.

Then import the object:

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

This object contains the following methods:

- `route(e: Endpoint[I, E, O, Any])(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(e: Endpoint[I, E, O, Any])(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(e: Endpoint[I, E, O, Any])(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(e: Endpoint[I, E, O, Any])(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]`.

Note that the second argument to `route` etc. is a function with one argument, 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`:

```
import sttp.tapir._
import sttp.tapir.server.vertx.VertxFutureServerInterpreter
import io.vertx.ext.web._
import scala.concurrent.Future

def logic(s: String, i: Int): Future[Either[Unit, String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Any] = ???
val aRoute: Router => Route = VertxFutureServerInterpreter().route(anEndpoint)((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:

```
import sttp.tapir._
import sttp.tapir.server.vertx.VertxFutureServerInterpreter
import sttp.tapir.server.vertx.VertxFutureServerInterpreter._
import io.vertx.core.Vertx
import io.vertx.ext.web._
import scala.concurrent.{Await, Future}
import scala.concurrent.duration._

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, Any] = ??? // your␣
↪definition here
```

(continues on next page)

(continued from previous page)

```
    def logic(s: String, i: Int): Future[Either[Unit, String]] = ??? // your logic␣
↪here
    val attach = VertxFutureServerInterpreter().route(anEndpoint)((logic _).tupled)
    attach(router) // your endpoint is now attached to the router, and the route has␣
↪been created
    Await.result(server.requestHandler(router).listen(9000).asScala, Duration.Inf)
  }
}
```

4.25.2 Configuration

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

4.25.3 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.

4.25.4 Cats Effect typeclasses

Add the following dependency

```
"com.softwaremill.sttp.tapir" %% "tapir-vertx-server" % "0.19.0-M9"
"com.softwaremill.sttp.shared" %% "fs2" % "LatestVersion"
```

to use this interpreter with Cats Effect typeclasses.

Then import the object:

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

This object contains the following methods:

- `route[F[_], I, E, O](e: Endpoint[I, E, O, Fs2Streams[F]])(logic: I => F[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.
- `routeRecoverErrors[F[_], I, E, O](e: Endpoint[I, E, O, Fs2Streams[F]])(logic: I => F[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.

Here is simple example which starts HTTP server with one route:

```
import cats.effect._
import cats.effect.std.Dispatcher
import io.vertx.core.Vertx
import io.vertx.ext.web.Router
import sttp.tapir._
import sttp.tapir.server.vertx.VertxCatsServerInterpreter
import sttp.tapir.server.vertx.VertxCatsServerInterpreter._
```

(continues on next page)

(continued from previous page)

```
object App extends IOApp {
  val responseEndpoint: Endpoint[String, Unit, String, Any] =
    endpoint
      .in("response")
      .in(query[String]("key"))
      .out(plainBody[String])

  def handler(req: String): IO[Either[Unit, String]] =
    IO.pure(Right(req))

  override def run(args: List[String]): IO[ExitCode] = {
    Dispatcher[IO]
      .flatMap { dispatcher =>
        Resource
          .make(
            IO.delay {
              val vertx = Vertx.vertx()
              val server = vertx.createHttpServer()
              val router = Router.router(vertx)
              val attach = VertxCatsServerInterpreter[IO](dispatcher).
↪route(responseEndpoint)(handler)
              attach(router)
              server.requestHandler(router).listen(8080)
            }.flatMap(_.asF[IO])
          )({ server =>
            IO.delay(server.close).flatMap(_.asF[IO].void)
          })
      }
      .use(_ => IO.never)
  }
}
```

This interpreter also supports streaming using FS2 streams:

```
import cats.effect._
import cats.effect.std.Dispatcher
import fs2._
import sttp.capabilities.fs2.Fs2Streams
import sttp.tapir._
import sttp.tapir.server.vertx.VertxCatsServerInterpreter

val streamedResponse =
  endpoint
    .in("stream")
    .in(query[Int]("key"))
    .out(streamTextBody(Fs2Streams[IO])(CodecFormat.TextPlain()))

def dispatcher: Dispatcher[IO] = ???

val attach = VertxCatsServerInterpreter(dispatcher).route(streamedResponse) { key =>
  IO.pure(Right(Stream.chunk(Chunk.array("Hello world!".getBytes)).repeatN(key)))
}
```

4.25.5 ZIO

Add the following dependency

```
"com.softwaremill.sttp.tapir" %% "tapir-vertx-server" % "0.19.0-M9"
"com.softwaremill.sttp.shared" %% "zio" % "LatestVersion"
```

to use this interpreter with ZIO.

Then import the object:

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

This object contains method `route[R, I, E, O](e: Endpoint[I, E, O, ZioStreams])(logic: I => ZIO[R, E, O])` which returns a function `Router => Route` that will create a route maching the endpoint definition, and attach your logic as an handler.

Here is simple example which starts HTTP server with one route:

```
import io.vertx.core.Vertx
import io.vertx.ext.web.Router
import sttp.tapir._
import sttp.tapir.server.vertx.VertxZioServerInterpreter
import sttp.tapir.server.vertx.VertxZioServerInterpreter._

import zio._

object Short extends zio.App {
  implicit val runtime = Runtime.default

  val responseEndpoint =
    endpoint
      .in("response")
      .in(query[String]("key"))
      .out(plainBody[String])

  val attach = VertxZioServerInterpreter().route(responseEndpoint) { key => UIO.
↪succeed(key) }

  override def run(args: List[String]): URIO[ZEnv, ExitCode] =
    ZManaged.make(ZIO.effect {
      val vertx = Vertx.vertx()
      val server = vertx.createHttpServer()
      val router = Router.router(vertx)
      attach(router)
      server.requestHandler(router).listen(8080)
    } >>= (_.asRIO))({ server =>
      ZIO.effect(server.close()).flatMap(_.asRIO).orDie
    }).useForever.as(ExitCode.success).orDie
}
```

This interpreter supports streaming using ZStreams.

4.26 Running as an zio-http server

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

```
"com.softwaremill.sttp.tapir" %% "tapir-zio-http" % "0.19.0-M9"
```

Now import the object:

```
import sttp.tapir.server.ziohttp.ZioHttpInterpreter
```

The `ZioHttpInterpreter` objects contains methods: `toHttp` and `toHttpRecoverErrors`.

The `toHttp` method requires the logic of the endpoint to be given as a function of type:

```
I => RIO[R, Either[E, O]]
```

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

For example:

```
import sttp.tapir._
import sttp.tapir.server.ziohttp.ZioHttpInterpreter
import zhttp.http.{Http, Request, Response}
import zio._

def countCharacters(s: String): RIO[Any, Either[Unit, Int]] =
  ZIO.succeed(Right(s.length))

val countCharactersEndpoint: Endpoint[String, Unit, Int, Any] =
  endpoint.in(stringBody).out(plainBody[Int])

val countCharactersHttp: Http[Any, Throwable, Request, Response[Any, Throwable]]  =
  ZioHttpInterpreter().toHttp(countCharactersEndpoint)(countCharacters)
```

Note that the second argument to `toHttp` is a function with one argument, 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`:

```
import sttp.tapir._
import sttp.tapir.server.ziohttp.ZioHttpInterpreter
import zhttp.http.{Http, Request, Response}
import zio._

def logic(s: String, i: Int): RIO[Any, Either[Unit,String]] = ???
val anEndpoint: Endpoint[(String, Int), Unit, String, Any] = ???
val anHttp: Http[Any, Throwable, Request, Response[Any, Throwable]] =
    ZioHttpInterpreter().toHttp(anEndpoint)((logic _).tupled)
```

4.26.1 Streaming

The zio-http interpreter accepts streaming bodies of type `Stream[Throwable, Byte]`, as described by the `ZioStreams` capability. Both response bodies and request bodies can be streamed. Usage: `streamBody(ZioStreams)(schema, format)`.

The capability can be added to the classpath independently of the interpreter through the `"com.softwaremill.sttp.shared" %% "zio"` dependency.

4.26.2 Configuration

The interpreter can be configured by providing an `ZioHttpServerOptions` value, see server options for details.

4.26.3 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.

4.27 Running using the AWS serverless stack

Tapir server endpoints can be packaged and deployed as an AWS Lambda function. To invoke the function, HTTP requests can be proxied through AWS API Gateway.

To configure API Gateway routes, and the Lambda function, tools like AWS SAM and Terraform can be used, to automate cloud deployments.

For an overview of how this works in more detail, see this blog post.

4.27.1 Serverless interpreters

To implement the Lambda function, a server interpreter is available, which takes tapir endpoints with associated server logic, and returns an `AwsRequest => F[AwsResponse]` function. This is used in the `AwsLambdaIORuntime` to implement the Lambda loop of reading the next request, computing and sending the response.

Currently, only an interpreter integrating with cats-effect is available (`AwsCatsEffectServerInterpreter`). To use, add the following dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-aws-lambda" % "0.19.0-M9"
```

To configure API Gateway and the Lambda function, you can use:

- the `AwsSamInterpreter` which interprets tapir `Endpoints` into an AWS SAM template file
- or the `AwsTerraformInterpreter` which interprets `Endpoints` into terraform configuration file.

Add one of the following dependencies:

```
"com.softwaremill.sttp.tapir" %% "tapir-aws-sam" % "0.19.0-M9"
"com.softwaremill.sttp.tapir" %% "tapir-aws-terraform" % "0.19.0-M9"
```

4.27.2 Examples

In our GitHub repository you'll find a `LambdaApiExample` handler which uses `AwsServerInterpreter` to route a hello endpoint along with `SamTemplateExample` and `TerraformConfigExample` which interpret endpoints to SAM/Terraform configuration. Go ahead and clone tapir project and select `project awsExamples` from sbt shell.

Make sure you have AWS command line tools installed.

SAM

To try it out using SAM template you don't need an AWS account.

- install AWS SAM command line tool
- run `assembly` task and `runMain sttp.tapir.serverless.aws.examples.SamTemplateExample`

- open a terminal and in tapir root directory run `sam local start-api --warm-containers EAGER`

That will create `template.yaml` and start up AWS Api Gateway locally. Hello endpoint will be available under `curl http://127.0.0.1:3000/api/hello`. First invocation will take a while but subsequent ones will be faster since the created container will be reused.

Terraform

To run the example using terraform you will need an AWS account, and an S3 bucket.

- install Terraform
- run `assembly` task
- open a terminal in `tapir/serverless/aws/examples/target/jvm-2.13` directory. That's where the fat jar is saved. You need to upload it into your s3 bucket. Using command line tools: `aws s3 cp tapir-aws-examples.jar s3://{your-bucket}/{your-key}`.
- Run `runMain sttp.tapir.serverless.aws.examples.TerraformConfigExample {your-aws-region} {your-bucket} {your-key}`
- open terminal in tapir root directory, run `terraform init` and `terraform apply`

That will create `api_gateway.tf.json` configuration and deploy Api Gateway and lambda function to AWS. Terraform will output the url of the created API Gateway which you can call followed by `/api/hello` path.

To destroy all the created resources run `terraform destroy`.

4.28 Server options

Each interpreter can be configured using an options object, which includes:

- how to create a file (when receiving a response that is mapped to a file, or when reading a file-mapped multipart part)
- if, and how to handle exceptions (see error handling)
- if, and how to log requests (see loggin & debugging)
- how to handle decode failures (see error handling)
- additional user-provided interceptors

To use custom server options pass them as an argument to the interpreter's `apply` method. For example, for `AkkaHttpServerOptions` and `AkkaHttpServerInterpreter`:

```
import sttp.tapir.server.interceptor.decodefailure.DecodeFailureHandler
import sttp.tapir.server.akkahttp.AkkaHttpServerOptions
import sttp.tapir.server.akkahttp.AkkaHttpServerInterpreter

val customDecodeFailureHandler: DecodeFailureHandler = ???

val customServerOptions: AkkaHttpServerOptions = AkkaHttpServerOptions
  .customInterceptors
  .decodeFailureHandler(customDecodeFailureHandler)
  .options

AkkaHttpServerInterpreter(customServerOptions)
```

4.28.1 Request interceptors

Request interceptors intercept whole request, and are called once for each request. They can provide additional endpoint interceptors, as well as modify the request, or the response.

The following request interceptors are provided by default (and if enabled, called in this order):

- the metrics interceptor, which by default is disabled
- the `RejectInterceptor`, which specifies what should be done when decoding the request has failed for all interpreted endpoints. The default is to return a 405 (method not allowed), if there's at least one decode failure on the method, and a "no-match" otherwise (which is handled in an intereprter-specific manner)

4.28.2 Endpoint interceptors

An `EndpointInterceptor` allows intercepting the handling of a request by an endpoint, when either the endpoint's inputs have been decoded successfully, or when decoding has failed.

The following interceptors are used by default, and if enabled, called in this order:

- exception interceptor
- logging interceptor
- decode failure handler interceptor

Note that while the request will be passed top-to-bottom, handling of the result will be done in opposite order. E.g., if the result is a failed effect (an exception), it will first be logged by the logging interceptor, and only later passed to the exception interceptor.

Using `customInterceptors` on the options companion object, it is possible to customise the built-in interceptors, as well as add new ones in-between the default initial (exception, logging) and decode failure interceptors. Customisation can include removing the interceptor altogether. Later, interceptors can be added using `.prependInterceptor` and `.appendInterceptor`.

4.29 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.

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

(continues on next page)

(continued from previous page)

```
val countCharactersServerEndpoint: ServerEndpoint[String, Unit, Int, Any, Future] =
  endpoint.in(stringBody).out(plainBody[Int]).serverLogic { s =>
    Future.successful[Either[Unit, Int]](Right(s.length))
  }

val countCharactersRoute: Route =
  AkkaHttpServerInterpreter().toRoute(countCharactersServerEndpoint)
```

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:

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

val endpoint1 = endpoint.in("hello").out(stringBody)
  .serverLogic { _ => Future.successful[Either[Unit, String]](Right("world")) }

val endpoint2 = endpoint.in("ping").out(stringBody)
  .serverLogic { _ => Future.successful[Either[Unit, String]](Right("pong")) }

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

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:

```
import sttp.tapir._
import scala.concurrent.Future

val echoEndpoint = endpoint
  .in(query[Int]("count"))
  .in(stringBody)
  .out(stringBody)
  .serverLogic { case (count, body) =>
     Future.successful[Either[Unit, String]](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:

```
import sttp.tapir._
import scala.concurrent.Future
```

(continues on next page)

(continued from previous page)

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

4.29.2 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:

```
import sttp.tapir._
import sttp.tapir.server._
import scala.concurrent.Future

implicit val ec = scala.concurrent.ExecutionContext.global

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[String, User, Unit, Int, Unit, Any, 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.successful[Either[Int, String]](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:

```
import sttp.tapir._

val secureHelloWorld2: Endpoint[(String, String), Int, String, Any] = 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):

```
import scala.concurrent.Future

case class User(name: String)
def auth(token: String): Future[Either[Int, User]] = ???

val secureHelloWorld2WithLogic = secureHelloWorld2
  .serverLogicPart(auth)
  .andThen { case (user, salutation) =>
    Future.successful[Either[Int, String]](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`.

4.29.3 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*.

4.30 Observability

Metrics collection is possible by creating `Metric` instances and adding them to server options via `MetricsInterceptor`. Certain endpoints can be ignored by adding their definitions to `ignoreEndpoints` list.

`Metric` wraps an aggregation object (like a counter or gauge), and needs to implement the `onRequest` function, which returns an `EndpointMetric` instance.

`Metric.onRequest` is used to create the proper metric description. Any additional data might be gathered there, like getting current timestamp and passing it down to `EndpointMetric` callbacks which are then executed in certain points of request processing.

There are three callbacks in `EndpointMetric`:

1. `onEndpointRequest` - called after a request matches an endpoint (inputs are successfully decoded), or when decoding inputs fails, but a downstream interceptor provides a response.
2. `onResponse` - called after response is assembled. Note that the response body might be lazily produced. To properly observe end-of-body, use the provided `BodyListener`.
3. `onException` - called after exception is thrown (in underlying streamed body, and/or on any other exception when there's no default response)

4.30.1 Labels

By default, request metrics are labeled by path and method. Response labels are additionally labelled by status code group. For example GET endpoint like `http://h:p/api/persons?name=Mike` returning 200 response will be labeled as `path="api/persons", method="GET", status="2xx"`. Query params are omitted by default, but it's possible to include them as shown in example below.

If the path contains captures, the label will include the path capture name instead of the actual value, e.g. `api/persons/{name}`.

Labels for default metrics can be customized, any attribute from `Endpoint`, `ServerRequest` and `ServerResponse` could be used, for example:

```
import sttp.tapir.metrics.MetricLabels

val labels = MetricLabels(
  forRequest = Seq(
    "path" -> { case (ep, _) => ep.renderPathTemplate() },
    "protocol" -> { case (_, req) => req.protocol }
```

(continues on next page)

(continued from previous page)

```
  ),
  forResponse = Seq()
)
```

4.30.2 Prometheus metrics

Add the following dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-prometheus-metrics" % "0.19.0-M9"
```

`PrometheusMetrics` encapsulates `CollectorReqistry` and `Metric` instances. It provides several ready to use metrics as well as an endpoint definition to read the metrics & expose them to the Prometheus server.

For example, using `AkkaServerInterpeter`:

```
import akka.http.scaladsl.server.Route
import io.prometheus.client.CollectorRegistry
import sttp.monad.FutureMonad
import sttp.tapir.metrics.prometheus.PrometheusMetrics
import sttp.tapir.server.akkahttp.{AkkaHttpServerInterpreter, AkkaHttpServerOptions}

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

implicit val monad: FutureMonad = new FutureMonad()

val prometheusMetrics = PrometheusMetrics[Future]("tapir", CollectorRegistry.
↪defaultRegistry)
  .withRequestsTotal()
  .withResponsesTotal()
  .withResponsesDuration()

val serverOptions: AkkaHttpServerOptions = AkkaHttpServerOptions
  .customInterceptors
  .metricsInterceptor(prometheusMetrics.metricsInterceptor())
  .options

val routes: Route = AkkaHttpServerInterpreter(serverOptions).
↪toRoute(prometheusMetrics.metricsEndpoint)
```

Custom metrics

Also, custom metric creation is possible and attaching it to `PrometheusMetrics`, for example:

```
import sttp.tapir.metrics.prometheus.PrometheusMetrics
import sttp.tapir.metrics.{EndpointMetric, Metric}
import io.prometheus.client.{CollectorRegistry, Counter}
import scala.concurrent.Future

// Metric for counting responses labeled by path, method and status code
val responsesTotal = Metric[Future, Counter](
  Counter
    .build()
    .namespace("tapir")
```

(continues on next page)

(continued from previous page)

```
    .name("responses_total")
    .help("HTTP responses")
    .labelNames("path", "method", "status")
    .register(CollectorRegistry.defaultRegistry),
  onRequest = { (req, counter, _) =>
    Future.successful(
      EndpointMetric()
        .onResponse { (ep, res) =>
          Future.successful {
            val path = ep.renderPathTemplate()
            val method = req.method.method
            val status = res.code.toString()
            counter.labels(path, method, status).inc()
          }
        }
    )
  }
)

val prometheusMetrics = PrometheusMetrics[Future]("tapir", CollectorRegistry.
↪defaultRegistry).withCustom(responsesTotal)
```

4.30.3 OpenTelemetry metrics

Add the following dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-opentelemetry-metrics" % "0.19.0-M9"
```

OpenTelemetry metrics are vendor-agnostic and can be exported using one of exporters from SDK.

`OpenTelemetryMetrics` encapsulates metric instances and needs a `Meter` from OpenTelemetry API to create default metrics, simply:

```
import sttp.tapir.metrics.opentelemetry.OpenTelemetryMetrics
import io.opentelemetry.api.metrics.{Meter, MeterProvider}
import scala.concurrent.Future

val provider: MeterProvider = ???
val meter: Meter = provider.get("instrumentation-name")

val metrics = OpenTelemetryMetrics[Future](meter)
  .withRequestsTotal()
  .withRequestsActive()
  .withResponsesTotal()
  .withResponsesDuration()

val metricsInterceptor = metrics.metricsInterceptor() // add me to your server options
```

4.31 Error handling

Error handling in tapir is divided into three areas:

1. Error outputs: defined per-endpoint, used for errors handled by the business logic

2. Failed effects: exceptions which are not handled by the server logic (corresponds to 5xx responses)
3. Decode failures: format errors, when the input values can't be decoded (corresponds to 4xx responses, or trying another endpoint)

While 1. is specific to an endpoint, handlers for 2. and 3. are typically the same for multiple endpoints, and are specified as part of the server's interpreter options.

4.31.1 Error outputs

Each endpoint can contain dedicated error outputs, in addition to outputs which are used in case of success. The business logic can then return either an error value, or a success value. Any business-logic-level errors should be signalled this way. This can include validation, failure of downstream services, or inability to serve the request at that time.

If the business logic signals errors as exceptions, some or all can be recovered from and mapped to an error value. For example:

```
import sttp.tapir._
import sttp.tapir.server.akkahttp.AkkaHttpServerInterpreter
import scala.concurrent.Future
import scala.util._

implicit val ec = scala.concurrent.ExecutionContext.global
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) =>
      println(s"Exception when running endpoint logic: $e")
      Success(Left(e.getMessage))
  }

AkkaHttpServerInterpreter().toRoute(
  endpoint
    .errorOut(plainBody[ErrorInfo])
    .out(plainBody[Int])
    .in(query[String]("name"))) {
  (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 (or similar for other interpreters).

4.31.2 Failed effects: unhandled exceptions

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 will be handled by the logging and exception interceptors. By default, an `ERROR` will be logged, and an `500 InternalServerError` returned.

4.31.3 Decode failures

Quite often user input will be malformed and decoding of the request 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:

- a `405 Method Not Allowed` is returned if multiple endpoints have been interpreted, and for at least one of them the path matched, but the method didn't. We assume that all endpoints for that path have been given to the interpreter, hence the response. This behavior can be customised or turned off using the `RejectInterceptor`
- 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.

The behavior described in the latter two points can be customised by providing a custom `sttp.tapir.server.DecodeFailureHandler` when creating the server options. This handler, 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 encountered for a specific endpoint is passed to the `DecodeFailureHandler`. Inputs are decoded in the following order: path, method, 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. Parts of the default behavior can be swapped, e.g. to return responses in a different format (other than plain text), or customise the error messages.

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`.

4.31.4 Customising how error messages are rendered

To return error responses in a different format (other than plain text), you can customise both the exception & decode failure handlers individually, or use the `CustomInterceptors.errorOutput` method which customises the default ones for you.

We'll need to provide both the endpoint output which should be used for error messages, along with the output's value:

```
import sttp.tapir._
import sttp.tapir.server.interceptor.ValuedEndpointOutput
import sttp.tapir.server.akkahttp.AkkaHttpServerOptions
import sttp.tapir.generic.auto._
import sttp.tapir.json.circe._
import io.circe.generic.auto._

case class MyFailure(msg: String)
def myFailureResponse(m: String): ValuedEndpointOutput[_] =
  ValuedEndpointOutput(jsonBody[MyFailure], MyFailure(m))

val myServerOptions: AkkaHttpServerOptions = AkkaHttpServerOptions
  .customInterceptors
  .errorOutput(myFailureResponse)
  .options
```

If you want to customise anything beyond the rendering of the error message, or use non-default implementations of the exception handler / decode failure handler, you'll still have to customise each by hand.

4.32 Logging & 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 providing an instance of the `ServerLog` class, which is part of server options.

The default implementation, `DefaultServerLog`, can be customised using the following flags:

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. `logExceptions`: 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.

4.33 Using as an sttp client

Add the dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-sttp-client" % "0.19.0-M9"
```

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

```
import sttp.tapir.client.sttp.SttpClientInterpreter
```

This objects contains two methods:

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

```
I => Request[Either[E, O], Any]
```

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

```
I => Request[DecodeResult[Either[E, O]], Any]
```

Note that the returned functions have one-argument: the input values 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.

4.33.1 Web sockets

To interpret a web socket endpoint, an additional streams-specific import is needed, so that the interpreter can convert sttp's `WebSocket` instance into a pipe. This logic is looked up via the `WebSocketToPipe` implicit.

The required imports are as follows:

```
import sttp.tapir.client.sttp.ws.akkahttp._ // for akka-streams // for akka-streams
import sttp.tapir.client.sttp.ws.fs2._      // for fs2
```

No additional dependencies are needed, as both of the above implementations are included in the main interpreter,] with dependencies on akka-streams and fs2 being marked as optional (hence these are not transitive).

4.33.2 Scala.JS

In this case add the following dependencies (note the `%%%` instead of the usual `%%`):

```
"com.softwaremill.sttp.tapir" %%% "tapir-sttp-client" % "0.19.0-M9"
"io.github.cquiroz" %%% "scala-java-time" % "2.2.0" // implementations of java.time␣
↪classes for Scala.JS
```

The client interpreter also supports Scala.JS, the request must then be sent using the sttp client Scala.JS Fetch backend.

You can check the `SttpClientTests` for a working example.

4.34 Using as a Play client

Add the dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-play-client" % "0.19.0-M9"
```

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

```
import sttp.tapir.client.play.PlayClientInterpreter
```

This objects contains two methods:

- `toRequestUnsafe(Endpoint, String)`: given the base URI returns a function, which will generate a request and a response parser which might throw an exception when decoding of the result fails

```
I => (StandaloneWSRequest, StandaloneWSResponse => Either[E, O])
```

- `toRequest(Endpoint, String)`: given the base URI returns a function, which will generate a request and a response parser which represents decoding errors as the `DecodeResult` class

```
I => (StandaloneWSRequest, StandaloneWSResponse => DecodeResult[Either[E, O]])
```

Note that the returned functions have one-argument: the input values 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, the two following are returned:

- a description of the request to be made, with the input value encoded as appropriate request parameters: path, query, headers and body. This can be further customised and sent using regular Play methods.
- a response parser to be applied to the response got after executing the request. The result will then contain the decoded error or success values (note that this can be the body enriched with data from headers/status code).

Example:

```
import sttp.tapir._
import sttp.tapir.client.play.PlayClientInterpreter
import sttp.capabilities.akka.AkkaStreams

import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

import play.api.libs.ws.StandaloneWSClient

def example[I, E, O, R >: AkkaStreams](implicit wsClient: StandaloneWSClient) {
  val e: Endpoint[I, E, O, R] = ???
  val inputArgs: I = ???

  val (req, responseParser) = PlayClientInterpreter()
      .toRequestUnsafe(e, s"http://localhost:9000")
      .apply(inputArgs)

  val result: Future[Either[E, O]] = req
      .execute()
      .map(responseParser)
}
```

4.34.1 Limitations

Multipart requests are not supported.

Streaming capabilities:

- only `AkkaStreams` is supported

4.35 Using as an http4s client

Add the dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-http4s-client" % "0.19.0-M9"
```

To interpret an endpoint definition as an `org.http4s.Request[F]`, import:

```
import sttp.tapir.client.http4s.Http4sClientInterpreter
```

This objects contains two methods:

- `toRequestUnsafe(Endpoint, Option[String])`: given the optional base URI, returns a function which generates a request and a response parser from endpoint inputs. Response parser throws an exception if decoding of the result fails.

  ```
  I => (org.http4s.Request[F], org.http4s.Response[F] => F[Either[E, O]])
  ```

- `toRequest(Endpoint, Option[String])`: given the optional base URI, returns a function which generates a request and a response parser from endpoint inputs. Response parser returns an instance of `DecodeResult` which contains the decoded response body or error details.

  ```
  I => (org.http4s.Request[F], org.http4s.Response[F] => F[DecodeResult[Either[E,␣
  ↪O]]])
  ```

Note that the returned functions have one-argument: the input values 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, the following values are returned:

- An instance of `org.http4s.Request[F]` with the input value encoded as appropriate request parameters: path, query, headers and body. The request can be further customised and sent using an http4s client, or run against `org.http4s.HttpRoutes[F]`.
- A response parser to be applied to the response received after executing the request. The result 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.

4.35.1 Limitations

- Multipart requests are not supported yet.
- WebSockets are not supported yet.
- Streaming capabilities:
  - only `Fs2Streams` are supported at the moment.

4.36 Generating OpenAPI documentation

To use, add the following dependencies:

```
"com.softwaremill.sttp.tapir" %% "tapir-openapi-docs" % "0.19.0-M9"
"com.softwaremill.sttp.tapir" %% "tapir-openapi-circe-yaml" % "0.19.0-M9"
```

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 `sttp.tapir.docs.openapi.OpenAPIDocsInterpreter` object:

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

val booksListing = endpoint.in(path[String]("bookId"))

val docs: OpenAPI = OpenAPIDocsInterpreter().toOpenAPI(booksListing, "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:

```
import sttp.tapir.openapi.Server

val docsWithServers: OpenAPI = OpenAPIDocsInterpreter().toOpenAPI(booksListing, "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 method on a list of endpoints:

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

The openapi case classes can then be serialised to YAML using Circe:

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

println(docs.toYaml)
```

Or to JSON:

```
import io.circe.Printer
import io.circe.syntax._
import sttp.tapir.openapi.circe._

println(Printer.spaces2.print(docs.asJson))
```

4.36.1 Options

Options can be customised by providing an instance of `OpenAPIDocsOptions` to the interpreter:

- `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.

- `defaultDecodeFailureOutput`: if an endpoint does not define a Bad Request response in `errorOut`, tapir will try to guess if decoding of inputs may fail, and add a 400 response if necessary. You can override this option to customize the mapping of endpoint's inputs to a default error response. If you'd like to disable this feature, just provide a function that always returns `None`:

```
OpenAPIDocsOptions.default.copy(defaultDecodeFailureOutput = _ => None)
```

4.36.2 Inlined and referenced schemas

All named schemas (that is, schemas which have the `Schema.name` property defined) will be referenced at point of use, and their definitions will be part of the `components` section. If you'd like a schema to be inlined, instead of referenced, *modify the schema* removing the name.

4.36.3 OpenAPI Specification Extensions

It's possible to extend specification with extensions. There are `.docsExtension` methods available on Input/Output parameters and on `endpoint`:

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

case class FruitAmount(fruit: String, amount: Int)

case class MyExtension(string: String, int: Int)

val sampleEndpoint =
  endpoint.post
    .in("path-hello" / path[String]("world").docsExtension("x-path", 22))
    .in(query[String]("hi").docsExtension("x-query", 33))
    .in(jsonBody[FruitAmount].docsExtension("x-request", MyExtension("a", 1)))
    .out(jsonBody[FruitAmount].docsExtension("x-response", List("array-0", "array-1
↪")).docsExtension("x-response", "foo"))
    .errorOut(stringBody.docsExtension("x-error", "error-extension"))
    .docsExtension("x-endpoint-level-string", "world")
    .docsExtension("x-endpoint-level-int", 11)
    .docsExtension("x-endpoint-obj", MyExtension("42.42", 42))

val rootExtensions = List(
  DocsExtension.of("x-root-bool", true),
  DocsExtension.of("x-root-list", List(1, 2, 4))
)

val openAPIYaml = OpenAPIDocsInterpreter().toOpenAPI(sampleEndpoint, Info("title", "1.
↪0"), rootExtensions).toYaml
```

However, to add extensions to other unusual places (like, `License` or `Server`, etc.) you should modify the `OpenAPI` object manually or using f.e. Quicklens

4.36.4 Exposing OpenAPI documentation

Exposing the OpenAPI can be done using Swagger UI or Redoc. The modules `tapir-swagger-ui` and `tapir-redoc` contain server endpoint definitions, which given the documentation in yaml format, will expose it using the given context path. To use, add as a dependency either:

```
"com.softwaremill.sttp.tapir" %% "tapir-swagger-ui" % "0.19.0-M9"
"com.softwaremill.sttp.tapir" %% "tapir-redoc" % "0.19.0-M9"
```

Then, you'll need to pass the server endpoints to your server interpreter. For example, using akka-http:

```
import sttp.tapir._
import sttp.tapir.docs.openapi.OpenAPIDocsInterpreter
import sttp.tapir.openapi.circe.yaml._
import sttp.tapir.server.akkahttp.AkkaHttpServerInterpreter
import sttp.tapir.swagger.SwaggerUI

import scala.concurrent.Future

val myEndpoints: Seq[Endpoint[_, _, _, _]] = ???
val docsAsYaml: String = OpenAPIDocsInterpreter().toOpenAPI(myEndpoints, "My App", "1.
↪0").toYaml

// add to your akka routes
val swaggerUIRoute = AkkaHttpServerInterpreter().
↪toRoute(SwaggerUI[Future](docsAsYaml))
```

Using with sbt-assembly

The `tapir-swagger-ui` modules rely on a file in the `META-INF` directory tree, to determine the version of the Swagger UI. You need to take additional measures if you package your application with sbt-assembly because the default merge strategy of the `assembly` task discards most artifacts in that directory. To avoid a `NullPointerException`, you need to include the following file explicitly:

```
assemblyMergeStrategy in assembly := {
  case PathList("META-INF", "maven", "org.webjars", "swagger-ui", "pom.properties") =>
    MergeStrategy.singleOrError
  case x =>
    val oldStrategy = (assemblyMergeStrategy in assembly).value
    oldStrategy(x)
}
```

4.37 Generating AsyncAPI documentation

To use, add the following dependencies:

```
"com.softwaremill.sttp.tapir" %% "tapir-asyncapi-docs" % "0.19.0-M9"
"com.softwaremill.sttp.tapir" %% "tapir-asyncapi-circe-yaml" % "0.19.0-M9"
```

Tapir contains a case class-based model of the asyncapi data structures in the `asyncapi/asyncapi-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 using the `sttp.tapir.docs.asyncapi.AsyncAPIInterpreter` object:

```
import sttp.capabilities.akka.AkkaStreams
import sttp.tapir._
import sttp.tapir.asyncapi.AsyncAPI
import sttp.tapir.docs.asyncapi.AsyncAPIInterpreter
import sttp.tapir.generic.auto._
import sttp.tapir.json.circe._
import io.circe.generic.auto._

case class Response(msg: String, count: Int)
val echoWS = endpoint.out(
  webSocketBody[String, CodecFormat.TextPlain, Response, CodecFormat.
↪Json](AkkaStreams))

val docs: AsyncAPI = AsyncAPIInterpreter().toAsyncAPI(echoWS, "Echo web socket", "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 `AsyncAPI` 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. Any servers provided to the `.toAsyncAPI` invocation will be supplemented with security requirements, as specified by the endpoints:

```
import sttp.tapir.asyncapi.Server

val docsWithServers: AsyncAPI = AsyncAPIInterpreter().toAsyncAPI(
  echoWS,
  "Echo web socket",
  "1.0",
  List("production" -> Server("api.example.com", "wss"))
)
```

Servers can also be later added through methods on the `AsyncAPI` object.

Multiple endpoints can be converted to an `AsyncAPI` instance by calling the method using a list of endpoints.

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

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

println(docs.toYaml)
```

4.37.1 Options

Options can be customised by providing an instance of `AsyncAPIDocsOptions` to the interpreter:

- `subscribeOperationId`: basing on the endpoint's path and the entire endpoint, determines the id of the subscribe operation. This can be later used by code generators as the name of the method to receive messages from the socket.
- `publishOperationId`: as above, but for publishing (sending messages to the web socket).

4.37.2 Inlined and referenced schemas

All named schemas (that is, schemas which have the `Schema.name` property defined) will be referenced at point of use, and their definitions will be part of the `components` section. If you'd like a schema to be inlined, instead of referenced, *modify the schema* removing the name.

4.37.3 AsyncAPI Specification Extensions

AsyncAPI supports adding extensions as well as OpenAPI. There is `docsExtension` method available on parameters and endpoints. There are `requestsDocsExtension` and `responsesDocsExtension` methods on `websocketBody`. Take a look at **OpenAPI Specification Extensions** section of *documentation* to get a feeling on how to use it.

4.37.4 Exposing AsyncAPI documentation

AsyncAPI documentation can be exposed through the AsyncAPI playground.

4.38 Testing

4.38.1 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" % "0.19.0-M9"
```

And the following imports:

```
import sttp.client3.testing.SttpBackendStub
import sttp.tapir.server.stub._
```

Given the following endpoint:

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

case class ResponseWrapper(value: Double)

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

Convert any endpoint to `SttpBackendStub`:

```
import sttp.client3.monad.IdMonad

val backend = SttpBackendStub
  .apply(IdMonad)
  .whenRequestMatchesEndpoint(e)
  .thenSuccess(ResponseWrapper(1.0))
```

A stub which executes an endpoint's server logic can also be created (here with an identity effect, but any supported effect can be used):

```
import sttp.client3.Identity
import sttp.client3.monad.IdMonad

val anotherBackend = SttpBackendStub
  .apply(IdMonad)
  .whenRequestMatchesEndpointThenLogic(e.serverLogic[Identity](_ =>␣
↪Right(ResponseWrapper(1.0))))
```

4.38.2 Black box testing

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

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.

Black box testing with mock-server integration

If you are writing integration tests for your application which communicates with some external systems(e.g payment providers, SMS providers, etc.), you could stub them using tapir's integration with mock-server

Add the following dependency:

```
"com.softwaremill.sttp.tapir" %% "tapir-sttp-mock-server" % "0.19.0-M9"
```

Imports:

```
import sttp.tapir.server.mockserver._
```

Then, given the following endpoint:

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

case class SampleIn(name: String, age: Int)

case class SampleOut(greeting: String)

val sampleJsonEndpoint = endpoint.post
  .in("api" / "v1" / "json")
  .in(header[String]("X-RequestId"))
  .in(jsonBody[SampleIn])
```

(continues on next page)

(continued from previous page)

```
  .errorOut(stringBody)
  .out(jsonBody[SampleOut])
```

and having any `SttpBackend` instance (for example, `TryHttpURLConnectionBackend` or with any other, arbitrary effect `F[_]` type), convert any endpoint to a **mock-server** expectation:

```
import sttp.client3.{TryHttpURLConnectionBackend, UriContext}

val testingBackend = TryHttpURLConnectionBackend()
val mockServerClient = SttpMockServerClient(baseUri = uri"http://localhost:1080",␣
↪testingBackend)

val in = "request-id-123" -> SampleIn("John", 23)
val out = SampleOut("Hello, John!")

val expectation = mockServerClient
  .whenInputMatches(sampleJsonEndpoint)(in)
  .thenSuccess(out)
  .get
```

Then you can try to send requests to the mock-server as you would do with live integration:

```
import sttp.tapir.client.sttp.SttpClientInterpreter
import sttp.client3.{TryHttpURLConnectionBackend, UriContext}

val testingBackend = TryHttpURLConnectionBackend()
val in = "request-id-123" -> SampleIn("John", 23)
val out = SampleOut("Hello, John!")

val result = SttpClientInterpreter()
  .toRequest(sampleJsonEndpoint, baseUri = Some(uri"http://localhost:1080"))
  .apply(in)
  .send(testingBackend)
  .get

result == out
```

4.38.3 Shadowed endpoints

It is possible to define a list of endpoints where some endpoints will be overlapping with each other. In such case when all matching requests will be handled by the first endpoint; the second endpoint will always be omitted. To detect such cases one can use `FindShadowedEndpoints` util class which takes an input of type `List[Endpoint[_, _, _, _]]` an outputs `Set[ShadowedEndpoint]`.

Example 1:

```
import sttp.tapir.testing.FindShadowedEndpoints

val e1 = endpoint.get.in("x" / paths)
val e2 = endpoint.get.in("x" / "y" / "x")
val e3 = endpoint.get.in("x")
val e4 = endpoint.get.in("y" / "x")
val res = FindShadowedEndpoints(List(e1, e2, e3, e4))
```

Results in:

```
res.toString
// res2: String = "Set(GET /x /y /x, is shadowed by: GET /x /..., GET /x, is shadowed␣
↪by: GET /x /...)"
```

Example 2:

```
import sttp.tapir.testing.FindShadowedEndpoints

val e1 = endpoint.get.in(path[String].name("y_1") / path[String].name("y_2"))
val e2 = endpoint.get.in(path[String].name("y_3") / path[String].name("y_4"))
val res = FindShadowedEndpoints(List(e1, e2))
```

Results in:

```
res.toString
// res3: String = "Set(GET /[y_3] /[y_4], is shadowed by: GET /[y_1] /[y_2])"
```

Note that the above takes into account only the method & the shape of the path. It does *not* take into account possible decoding failures: these might impact request-endpoint matching, and the exact behavior is determined by the `DecodeFailureHandler` used.

4.39 Generate endpoint definitions from an OpenAPI YAML

Note: This is a really early alpha implementation.

4.39.1 Installation steps

Add the sbt plugin to the `project/plugins.sbt`:

```
addSbtPlugin("com.softwaremill.sttp.tapir" % "sbt-openapi-codegen" % "0.19.0-M9")
```

Enable the plugin for your project in the `build.sbt`:

```
enablePlugins(OpenapiCodegenPlugin)
```

Add your OpenApi file to the project, and override the `openapiSwaggerFile` setting in the `build.sbt`:

```
openapiSwaggerFile := baseDirectory.value / "swagger.yaml"
```

At this point your compile step will try to generate the endpoint definitions to the `sttp.tapir.generated.TapirGeneratedEndpoints` object, where you can access the defined case-classes and endpoint definitions.

4.39.2 Usage and options

The generator currently supports these settings, you can override them in the `build.sbt`;

setting	default value	description
openapiSwaggerFile	baseDirectory.value / “swagger.yaml”	The swagger file with the api definitions.
openapiPackage	sttp.tapir.generated	The name for the generated package.
openapiObject	TapirGeneratedEndpoints	The name for the generated object.

The general usage is;

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

val docs = TapirGeneratedEndpoints.generatedEndpoints.toOpenAPI("My Bookshop", "1.0")
```

Limitations

Currently, the generated code depends on `"io.circe" %% "circe-generic"`. In the future probably we will make the encoder/decoder json lib configurable (PRs welcome).

We currently miss a lot of OpenApi features like:

- tags
- enums/ADTs
- missing model types and meta descriptions (like date, minLength)
- file handling

4.40 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.

4.40.1 GraphQL

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

4.40.2 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.

4.41 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 AkkaHttpServerInterpreter
  with SttpClientInterpreter
  with OpenAPIDocsInterpreter
  with SchemaDerivation
  with TapirJsonCirce
  with TapirOpenAPICirceYaml
  with TapirAliases
```

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

4.42 Troubleshooting

4.42.1 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
```

4.42.2 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
```

4.43 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.

4.43.1 Acknowledgments

Tuple-concatenating code is copied from akka-http

Generic derivation configuration is copied from circe