sttp Documentation

Release 3

Adam Warski

Feb 18, 2021

Getting started

Welcome!

sttp client is an open-source library which provides a clean, programmer-friendly API to describe HTTP requests and how to handle responses. Requests are sent using one of the backends, which wrap other Scala or Java HTTP client implementations. The backends can integrate with a variety of Scala stacks, providing both synchronous and asynchronous, procedural and functional interfaces.

Backend implementations include ones based on akka-http, async-http-client, http4s, OkHttp, and HTTP clients which ship with Java. They integrate with Akka, Monix, fs2, cats-effect, scalaz and ZIO. Supported Scala versions include 2.11, 2.12, 2.13 and 3.

Here's a quick example of sttp client in action:

```
import sttp.client3._

val query = "http language:scala"
val sort: Option[String] = None

// the `query` parameter is automatically url-encoded
// `sort` is removed, as the value is not defined
val request = basicRequest.get(
  uri"https://api.github.com/search/repositories?q=$query&sort=$sort")

val backend = HttpURLConnectionBackend()
val response = request.send(backend)

// response.header(...): Option[String]
println(response.header("Content-Length"))

// response.body: by default read into an Either[String, String]
// to indicate failure or success
println(response.body)

// alternatively, if you prefer to pass the backend explicitly, instead
// of using implicits, you can also call:
val sameResponse = backend.send(request)
```

For more examples, see the *usage examples* section. To start using sttp client in your project, see the *quickstart*. Or, browse the documentation to find the topics that interest you the most!

CHAPTER 1

Other sttp projects

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

- sttp client: this project
- sttp tapir: Typed API descRiptions
- sttp model: simple HTTP model classes (used by client & tapir)
- sttp shared: shared web socket, FP abstractions, capabilities and streaming code.

Third party projects:

- sttp-oauth2: OAuth2 client library for Scala

CHAPTER 2

Sponsors

Development and maintenance of sttp client 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 3

Try sttp client in your browser!

CHAPTER 4

Table of contents

4.1 Quickstart

The core sttp client API comes in a single jar, with a transitive dependency on sttp model. This also includes a default, *synchronous* backend, which is based on Java's `HttpURLConnection`.

To integrate with other parts of your application, you'll often need to use an alternate backend (but what's important is that the API remains the same!). See the section on *backends* for a short guide on which backend to choose, and a list of all implementations.

4.1.1 Using sbt

The basic dependency which provides the API and the default synchronous backend is:

```
"com.softwaremill.sttp.client3" %% "core" % "3.1.3"
```

`sttp client` is available for Scala 2.11, 2.12 and 2.13, and requires Java 8, as well as for Scala 3.

`sttp client` is also available for Scala.js 1.0. Note that not all modules are compatible and there are no backends that can be used on both. The last version compatible with Scala.js 0.6 was 2.2.1. Scala Native is supported as well.

4.1.2 Using Ammonite

If you are an Ammonite user, you can quickly start experimenting with sttp by copy-pasting the following:

```
import $ivy.`com.softwaremill.sttp.client3::core:3.1.3`
import sttp.client3.quick._
quickRequest.get(uri"http://httpbin.org/ip").send(backend)
```

Importing the `quick` object has the same effect as importing `sttp.client3._`, plus defining a synchronous backend (`implict val backend = HttpURLConnectionBackend()`), so that sttp can be used right away.

If the default `HttpURLConnectionBackend` for some reason is insufficient, you can also use one based on OkHttp or HttpClient:

```
import $ivy.`com.softwaremill.sttp.client3::okhttp-backend:3.1.3`
import sttp.client3.okhttp.quick._
quickRequest.get(uri"http://httpbin.org/ip").send(backend)
```

4.1.3 Imports

Working with sttp is most convenient if you import the `sttp.client3` package entirely:

```
import sttp.client3._
```

This brings into scope the starting point for defining requests and some helper methods. All examples in this guide assume that this import is in place.

And that's all you need to start using sttp client! To create and send your first request, import the above, type `basicRequest.` and see where your IDE's auto-complete gets you! Here's a simple request, using the synchronous backend:

```
import sttp.client3._

val backend = HttpURLConnectionBackend()
val response = basicRequest
  .body("Hello, world!")
  .post(uri"https://httpbin.org/post?hello=world").send(backend)

println(response.body)
```

Next, read on about the *how sttp client works* or see some *examples*.

4.2 How sttp client works

4.2.1 Describe the request

This first step when using sttp client is describing the request that you'd like to send.

A request is represented as an immutable data structure of type `RequestT` (as in Request Template). The basic request is provided as the `basicRequest` value, in the `sttp.client3` package. It can be refined using one of the available methods, such as `.header`, `.body`, `.get(Uri)`, `.responseAs`, etc.

A `RequestT` value contains both information on what to include in the request, but also how to handle the response body.

To start describing a request, import the sttp client package and customise `basicRequest`:

```
import sttp.client3._
val myRequest: Request[_, _] = ??? // basicRequest.(...)
```

An alternative to importing the `sttp.client3._` package, is to extend the `sttp.client3.SttpApi` trait. That way, multiple integrations can be grouped in one object, thus reducing the number of necessary imports.

4.2.2 Send the request

Once the request is described as a value, it can be sent. To send a request, you'll need an `SttpBackend`.

The backend is where most of the work happens: the request is translated to a backend-specific form; a connection is opened, data sent and received; finally, the backend-specific response is translated to sttp's `Response`, as described in the request.

A backend can be synchronous, that is, sending a request can be a blocking operation. When invoking `myRequest.send(backend)`, you'll get a value of type `Response[T]`. Backends can also be asynchronous, and evaluate the send operation eagerly or lazily. For example, when using the *Akka backend*, `myRequest.send(backend)` will return a `Future[Response[T]]`: an eagerly-evaluated, asynchronous result. When using a *Monix backend*, you'll get back a `Task[Response[T]]`: a lazily-evaluated, but also non-blocking and asynchronous result.

Backends manage the connection pool, thread pools for handling responses, depending on the implementation provide various configuration options, and optionally support *streaming* and *websockets*. They typically need to be created upon application startup, and closed when the application terminates.

For example, the following sends a synchronous request, using the default JVM backend:

```
import sttp.client3._
val myRequest: Request[String, Any] = ???
val backend = HttpURLConnectionBackend()
val response = myRequest.send(backend)
```

4.2.3 Next steps

Read more about:

- *describing the request*
- the *RequestT type*
- specifying how to handle the *response body*
- *available backends*

4.3 Goals of the project

- provide a simple, discoverable, no-surprises, reasonably type-safe API for making HTTP requests and reading responses
- separate definition of a request from request execution
- provide immutable, easily modifiable data structures for requests and responses
- support multiple execution backends, both synchronous and asynchronous
- provide support for backend-specific request/response streaming
- minimum dependencies

See also the blog posts:

- Introduction to sttp
- sttp streaming & URI interpolators
- sttp2: an overview of proposed changes

- Migrating to sttp client 2.x and tapir 0.12.x
- What's coming up in sttp client 3?

4.3.1 Non-goals of the project

- implement a full HTTP client. Instead, sttp client wraps existing HTTP clients, providing a consistent, programmer-friendly API. All network-related concerns such as sending the requests, connection pooling, receiving responses are delegated to the chosen backend
- provide ultimate flexibility in defining the request. While it's possible to define *most* valid HTTP requests, e.g. some of the less common body chunking approaches aren't available

4.3.2 How is sttp different from other libraries?

- immutable request builder which doesn't impose any order in which request parameters need to be specified. Such an approach allows defining partial requests with common cookies/headers/options, which can later be specialized using a specific URI and HTTP method.
- support for multiple backends, both synchronous and asynchronous, with backend-specific streaming support
- URI interpolator with context-aware escaping, optional parameters support and parameter collections
- description of how to handle the response is combined with the description of the request to send

4.4 Community

If you have a question, or hit a problem, feel free to ask on our gitter channel!

Or, if you encounter a bug, something is unclear in the code or documentation, don't hesitate and open an issue on GitHub.

We are also always looking for contributions and new ideas, so if you'd like to get into the project, check out the open issues, or post your own suggestions!

4.5 Usage examples

All of the examples are available in the sources in runnable form.

4.5.1 POST a form using the synchronous backend

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "core" % "3.1.3")
```

Example code:

```
package sttp.client3.examples

object PostFormSynchronous extends App {
  import sttp.client3._
```

(continues on next page)

(continued from previous page)

```
  val signup = Some("yes")

  val request = basicRequest
    // send the body as form data (x-www-form-urlencoded)
    .body(Map("name" -> "John", "surname" -> "doe"))
    // use an optional parameter in the URI
    .post(uri"https://httpbin.org/post?signup=$signup")

  val backend = HttpURLConnectionBackend()
  val response = request.send(backend)

  println(response.body)
  println(response.headers)
}
```

4.5.2 GET and parse JSON using the akka-http backend and json4s

Required dependencies:

```
libraryDependencies ++= List(
  "com.softwaremill.sttp.client3" %% "akka-http-backend" % "3.1.3",
  "com.softwaremill.sttp.client3" %% "json4s" % "3.1.3",
  "org.json4s" %% "json4s-native" % "3.6.0"
)
```

Example code:

```
package sttp.client3.examples

object GetAndParseJsonAkkaHttpJson4s extends App {
  import scala.concurrent.Future

  import sttp.client3._
  import sttp.client3.akkahttp._
  import sttp.client3.json4s._

  import scala.concurrent.ExecutionContext.Implicits.global

  case class HttpBinResponse(origin: String, headers: Map[String, String])

  implicit val serialization = org.json4s.native.Serialization
  implicit val formats = org.json4s.DefaultFormats
  val request = basicRequest
    .get(uri"https://httpbin.org/get")
    .response(asJson[HttpBinResponse])

  val backend: SttpBackend[Future, Any] = AkkaHttpBackend()
  val response: Future[Response[Either[ResponseException[String, Exception],␣
↪HttpBinResponse]]] =
    request.send(backend)

  for {
    r <- response
  } {
    println(s"Got response code: ${r.code}")
```

(continues on next page)

(continued from previous page)

```
    println(r.body)
    backend.close()
  }
}
```

4.5.3 GET and parse JSON using the ZIO async-http-client backend and circe

Required dependencies:

```
libraryDependencies ++= List(
  "com.softwaremill.sttp.client3" %% "async-http-client-backend-zio" % "3.1.3",
  "com.softwaremill.sttp.client3" %% "circe" % "3.1.3",
  "io.circe" %% "circe-generic" % "0.13.0"
)
```

Example code:

```
package sttp.client3.examples

import sttp.client3._
import sttp.client3.circe._
import sttp.client3.asynchttpclient.zio._
import io.circe.generic.auto._
import zio._
import zio.console.Console

object GetAndParseJsonZioCirce extends App {

  override def run(args: List[String]): ZIO[ZEnv, Nothing, ExitCode] = {

    case class HttpBinResponse(origin: String, headers: Map[String, String])

    val request = basicRequest
      .get(uri"https://httpbin.org/get")
      .response(asJson[HttpBinResponse])

    // create a description of a program, which requires two dependencies in the␣
↪environment:
    // the SttpClient, and the Console
    val sendAndPrint: ZIO[Console with SttpClient, Throwable, Unit] = for {
      response <- send(request)
      _ <- console.putStrLn(s"Got response code: ${response.code}")
      _ <- console.putStrLn(response.body.toString)
    } yield ()

    // provide an implementation for the SttpClient dependency; other dependencies are
    // provided by Zio
    sendAndPrint
      .provideCustomLayer(AsyncHttpClientZioBackend.layer())
      .exitCode
  }
}
```

4.5.4 GET and parse JSON using the async-http-client Monix backend and circe, treating deserialization errors as failed effects

Required dependencies:

```
libraryDependencies ++= List(
  "com.softwaremill.sttp.client3" %% "async-http-client-backend-monix" % "3.1.3",
  "com.softwaremill.sttp.client3" %% "circe" % "3.1.3",
  "io.circe" %% "circe-generic" % "0.13.0"
)
```

Example code:

```
package sttp.client3.examples

import io.circe.generic.auto._
import sttp.client3._
import sttp.client3.asynchttpclient.monix.AsyncHttpClientMonixBackend
import sttp.client3.circe._

object GetAndParseJsonGetRightMonixCirce extends App {
  import monix.execution.Scheduler.Implicits.global

  case class HttpBinResponse(origin: String, headers: Map[String, String])

  val request: Request[HttpBinResponse, Any] = basicRequest
    .get(uri"https://httpbin.org/get")
    .response(asJson[HttpBinResponse].getRight)

  AsyncHttpClientMonixBackend
    .resource()
    .use { backend =>
      request.send(backend).map { response: Response[HttpBinResponse] =>
        println(s"Got response code: ${response.code}")
        println(response.body)
      }
    }
    .runSyncUnsafe()
}
```

4.5.5 Log requests & responses using slf4j

Required dependencies:

```
libraryDependencies ++= List(
  "com.softwaremill.sttp.client3" %% "slf4j-backend" % "3.1.3",
  "com.softwaremill.sttp.client3" %% "circe" % "3.1.3",
  "io.circe" %% "circe-generic" % "0.13.0"
)
```

Example code:

```
package sttp.client3.examples

import io.circe.generic.auto._
import sttp.client3._
```

(continues on next page)

(continued from previous page)

```
import sttp.client3.circe._
import sttp.client3.logging.slf4j.Slf4jLoggingBackend

object LogRequestsSlf4j extends App {
  case class HttpBinResponse(origin: String, headers: Map[String, String])

  val request = basicRequest
    .get(uri"https://httpbin.org/get")
    .response(asJson[HttpBinResponse].getRight)

  val backend: SttpBackend[Identity, Any] =
    Slf4jLoggingBackend(
      HttpURLConnectionBackend(),
      includeTiming = true,
      logRequestBody = false,
      logResponseBody = false
    )

  try {
    val response: Response[HttpBinResponse] = request.send(backend)
    println("Done! " + response.code)
  } finally backend.close()
}
```

4.5.6 POST and serialize JSON using the Monix async-http-client backend and circe

Required dependencies:

```
libraryDependencies ++= List(
  "com.softwaremill.sttp.client3" %% "async-http-client-backend-monix" % "3.1.3",
  "com.softwaremill.sttp.client3" %% "circe" % "3.1.3",
  "io.circe" %% "circe-generic" % "0.13.0"
)
```

Example code:

```
package sttp.client3.examples

object PostSerializeJsonMonixAsyncHttpClientCirce extends App {
  import sttp.client3._
  import sttp.client3.circe._
  import sttp.client3.asynchttpclient.monix._
  import io.circe.generic.auto._
  import monix.eval.Task

  case class Info(x: Int, y: String)

  val postTask = AsyncHttpClientMonixBackend().flatMap { backend =>
    val r = basicRequest
      .body(Info(91, "abc"))
      .post(uri"https://httpbin.org/post")

    r.send(backend)
      .flatMap { response => Task(println(s"""Got ${response.code} response, body:\n$
↪{response.body}""")) }
```

(continues on next page)

(continued from previous page)

```
      .guarantee(backend.close())
  }

  import monix.execution.Scheduler.Implicits.global
  postTask.runSyncUnsafe()
}
```

4.5.7 Test an endpoint which requires multiple query parameters

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "core" % "3.1.3")
```

Example code:

```
package sttp.client3.examples

object TestEndpointMultipleQueryParameters extends App {
  import sttp.client3._
  import sttp.client3.testing._

  val backend = SttpBackendStub.synchronous
    .whenRequestMatches(_.uri.paramsMap.contains("filter"))
    .thenRespond("Filtered")
    .whenRequestMatches(_.uri.path.contains("secret"))
    .thenRespond("42")

  val parameters1 = Map("filter" -> "name=mary", "sort" -> "asc")
  println(
    basicRequest
      .get(uri"http://example.org?search=true&$parameters1")
      .send(backend)
      .body
  )

  val parameters2 = Map("sort" -> "desc")
  println(
    basicRequest
      .get(uri"http://example.org/secret/read?$parameters2")
      .send(backend)
      .body
  )
}
```

4.5.8 Open a websocket using ZIO

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "async-http-client-
↪backend-zio" % "3.1.3")
```

Example code:

```
package sttp.client3.examples

import sttp.client3._
import sttp.client3.asynchttpclient.zio._
import sttp.ws.WebSocket
import zio._
import zio.console.Console

object WebSocketZio extends App {
  def useWebSocket(ws: WebSocket[RIO[Console, *]]): RIO[Console, Unit] = {
    def send(i: Int) = ws.sendText(s"Hello $i!")
    val receive = ws.receiveText().flatMap(t => console.putStrLn(s"RECEIVED: $t"))
    send(1) *> send(2) *> receive *> receive
  }

  // create a description of a program, which requires two dependencies in the␣
↪environment:
  // the SttpClient, and the Console
  val sendAndPrint: RIO[Console with SttpClient, Response[Unit]] =
    sendR(basicRequest.get(uri"wss://echo.websocket.org").
↪response(asWebSocketAlways(useWebSocket)))

  override def run(args: List[String]): ZIO[ZEnv, Nothing, ExitCode] = {
    // provide an implementation for the SttpClient dependency; other dependencies are
    // provided by Zio
    sendAndPrint
      .provideCustomLayer(AsyncHttpClientZioBackend.layer())
      .exitCode
  }
}
```

4.5.9 Open a websocket using FS2 streams

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "async-http-client-
↪backend-fs2 % "3.1.3")
```

Example code:

```
package sttp.client3.examples

import cats.effect.{Blocker, ContextShift, IO}
import fs2._
import sttp.capabilities.fs2.Fs2Streams
import sttp.client3._
import sttp.client3.asynchttpclient.fs2.AsyncHttpClientFs2Backend
import sttp.ws.WebSocketFrame

import scala.concurrent.ExecutionContext.global

object WebSocketStreamFs2 extends App {
  implicit val cs: ContextShift[IO] = IO.contextShift(scala.concurrent.
↪ExecutionContext.global)

  def webSocketFramePipe: Pipe[IO, WebSocketFrame.Data[_], WebSocketFrame] = { input␣
↪=>
```

(continues on next page)

(continued from previous page)

```
    Stream.emit(WebSocketFrame.text("1")) ++ input.flatMap {
      case WebSocketFrame.Text("10", _, _) =>
        println("Received 10 messages, sending close frame")
        Stream.emit(WebSocketFrame.close)
      case WebSocketFrame.Text(n, _, _) =>
        println(s"Received $n messages, replying with $n+1")
        Stream.emit(WebSocketFrame.text((n.toInt + 1).toString))
      case _ => Stream.empty // ignoring
    }
  }

  AsyncHttpClientFs2Backend
    .resource[IO](Blocker.liftExecutionContext(global))
    .use { backend =>
      basicRequest
        .response(asWebSocketStream(Fs2Streams[IO])(webSocketFramePipe))
        .get(uri"wss://echo.websocket.org")
        .send(backend)
        .void
    }
    .unsafeRunSync()
}
```

4.5.10 Test Monix websockets

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "async-http-client-
↪backend-monix % "3.1.3")
```

Example code:

```
package sttp.client3.examples

import monix.eval.Task
import sttp.capabilities.WebSockets
import sttp.capabilities.monix.MonixStreams
import sttp.client3._
import sttp.client3.asynchttpclient.monix.AsyncHttpClientMonixBackend
import sttp.client3.testing.SttpBackendStub
import sttp.model.StatusCode
import sttp.ws.{WebSocket, WebSocketFrame}
import sttp.ws.testing.WebSocketStub

object WebSocketTesting extends App {
  // the web socket-handling logic
  def useWebSocket(ws: WebSocket[Task]): Task[Unit] = {
    def send(i: Int) = ws.sendText(s"Hello $i!")
    val receive = ws.receiveText().flatMap(t => Task(println(s"RECEIVED [$t]")))
    send(1) *> send(2) *> receive *> receive
  }

  // the request description
  def openWebSocket(backend: SttpBackend[Task, WebSockets]): Task[Unit] = {
    basicRequest
```

(continues on next page)

(continued from previous page)

```
      .response(asWebSocket(useWebSocket))
      .get(uri"wss://echo.websocket.org")
      .send(backend)
      .void
  }

  // the backend stub which we'll use instead of a "real" backend
  val stubBackend: SttpBackendStub[Task, MonixStreams with WebSockets] =
    AsyncHttpClientMonixBackend.stub
      .whenRequestMatches(_.uri.toString().contains("echo.websocket.org"))
      .thenRespond(
        WebSocketStub.noInitialReceive.thenRespond {
          case WebSocketFrame.Text(payload, _, _) =>
            List(WebSocketFrame.text(s"response to: $payload"))
          case _ => Nil // ignoring other types of messages
        },
        StatusCode.SwitchingProtocols
      )

  // running the test
  import monix.execution.Scheduler.Implicits.global
  openWebSocket(stubBackend).runSyncUnsafe()
}
```

4.5.11 Open a websocket using Akka

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "akka-http-backend" %
↪"3.1.3")
```

Example code:

```
package sttp.client3.examples

import sttp.client3._
import sttp.client3.akkahttp.AkkaHttpBackend
import sttp.ws.WebSocket

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

object WebSocketAkka extends App {
  def useWebSocket(ws: WebSocket[Future]): Future[Unit] = {
    def send(i: Int) = ws.sendText(s"Hello $i!")
    def receive() = ws.receiveText().map(t => println(s"RECEIVED: $t"))
    for {
      _ <- send(1)
      _ <- send(2)
      _ <- receive()
      _ <- receive()
    } yield ()
  }

  val backend = AkkaHttpBackend()
```

(continues on next page)

(continued from previous page)

```
  basicRequest
    .response(asWebSocket(useWebSocket))
    .get(uri"wss://echo.websocket.org")
    .send(backend)
    .onComplete(_ => backend.close())
}
```

4.5.12 Open a websocket using Monix

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "async-http-client-
↪backend-monix" % "3.1.3")
```

Example code:

```
package sttp.client3.examples

import monix.eval.Task
import sttp.client3._
import sttp.client3.asynchttpclient.monix.AsyncHttpClientMonixBackend
import sttp.ws.WebSocket

object WebSocketMonix extends App {
  import monix.execution.Scheduler.Implicits.global

  def useWebSocket(ws: WebSocket[Task]): Task[Unit] = {
    def send(i: Int) = ws.sendText(s"Hello $i!")
    val receive = ws.receiveText().flatMap(t => Task(println(s"RECEIVED: $t")))
    send(1) *> send(2) *> receive *> receive
  }

  AsyncHttpClientMonixBackend
    .resource()
    .use { backend =>
      basicRequest
        .response(asWebSocket(useWebSocket))
        .get(uri"wss://echo.websocket.org")
        .send(backend)
        .void
    }
    .runSyncUnsafe()
}
```

4.5.13 Stream request and response bodies using fs2

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "async-http-client-
↪backend-fs2" % "3.1.3")
```

Example code:

```
package sttp.client3.examples

import sttp.client3._
import sttp.client3.asynchttpclient.fs2.AsyncHttpClientFs2Backend
import cats.effect.{Blocker, ContextShift, IO}
import cats.instances.string._
import fs2.{Stream, text}
import sttp.capabilities.fs2.Fs2Streams

import scala.concurrent.ExecutionContext.global

object StreamFs2 extends App {
  implicit val cs: ContextShift[IO] = IO.contextShift(scala.concurrent.
↪ExecutionContext.global)

  def streamRequestBody(backend: SttpBackend[IO, Fs2Streams[IO]]): IO[Unit] = {
    val stream: Stream[IO, Byte] = Stream.emits("Hello, world".getBytes)

    basicRequest
      .streamBody(Fs2Streams[IO])(stream)
      .post(uri"https://httpbin.org/post")
      .send(backend)
      .map { response => println(s"RECEIVED:\n${response.body}") }
  }

  def streamResponseBody(backend: SttpBackend[IO, Fs2Streams[IO]]): IO[Unit] = {
    basicRequest
      .body("I want a stream!")
      .post(uri"https://httpbin.org/post")
      .response(asStreamAlways(Fs2Streams[IO])(_.chunks.through(text.utf8DecodeC).
↪compile.foldMonoid))
      .send(backend)
      .map { response => println(s"RECEIVED:\n${response.body}") }
  }

  val effect = AsyncHttpClientFs2Backend.resource[IO](Blocker.
↪liftExecutionContext(global)).use { backend =>
    streamRequestBody(backend).flatMap(_ => streamResponseBody(backend))
  }

  effect.unsafeRunSync()
}
```

4.5.14 Stream request and response bodies using zio-stream

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "async-http-client-
↪backend-zio" % "3.1.3")
```

Example code:

```
package sttp.client3.examples

import sttp.capabilities.zio.ZioStreams
import sttp.client3._
```

(continues on next page)

(continued from previous page)

```
import sttp.client3.asynchttpclient.zio.{AsyncHttpClientZioBackend, SttpClient, send}
import zio._
import zio.console._
import zio.stream._

object StreamZio extends App {
  def streamRequestBody: RIO[Console with SttpClient, Unit] = {
    val stream: Stream[Throwable, Byte] = Stream("Hello, world".getBytes: _*)

    send(
      basicRequest
        .streamBody(ZioStreams)(stream)
        .post(uri"https://httpbin.org/post")
    ).flatMap { response => putStrLn(s"RECEIVED:\n${response.body}") }
  }

  def streamResponseBody: RIO[Console with SttpClient, Unit] = {
    send(
      basicRequest
        .body("I want a stream!")
        .post(uri"https://httpbin.org/post")
        .response(asStreamAlways(ZioStreams)(_.transduce(Transducer.utf8Decode).fold("
↪")(_ + _)))
    ).flatMap { response => putStrLn(s"RECEIVED:\n${response.body}") }
  }

  override def run(args: List[String]): ZIO[ZEnv, Nothing, ExitCode] = {
    (streamRequestBody *> streamResponseBody)
      .provideCustomLayer(AsyncHttpClientZioBackend.layer())
      .exitCode
  }
}
```

4.5.15 Retry a request using ZIO

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "async-http-client-
↪backend-zio" % "3.1.3")
```

Example code:

```
package sttp.client3.examples

import sttp.client3._
import sttp.client3.asynchttpclient.zio.AsyncHttpClientZioBackend
import zio.{ExitCode, Schedule, ZIO}
import zio.clock.Clock
import zio.duration._

object RetryZio extends zio.App {
  override def run(args: List[String]): ZIO[zio.ZEnv, Nothing, ExitCode] = {
    AsyncHttpClientZioBackend().flatMap { backend =>
      val localhostRequest = basicRequest
        .get(uri"http://localhost/test")
```

(continues on next page)

(continued from previous page)

```
        .response(asStringAlways)

      val sendWithRetries: ZIO[Clock, Throwable, Response[String]] = localhostRequest
        .send(backend)
        .either
        .repeat(
          Schedule.spaced(1.second) *>
            Schedule.recurs(10) *>
            Schedule.recurWhile(result => RetryWhen.Default(localhostRequest, result))
        )
        .absolve

      sendWithRetries.ensuring(backend.close().ignore)
    }.exitCode
  }
}
```

4.5.16 GET parsed and raw response bodies

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client3" %% "core" % "3.1.3")
```

Example code:

```
package sttp.client3.examples

import io.circe
import io.circe.generic.auto._
import sttp.client3._
import sttp.client3.circe._

object GetRawResponseBodySynchronous extends App {
  case class HttpBinResponse(origin: String, headers: Map[String, String])

  val request = basicRequest
    .get(uri"https://httpbin.org/get")
    .response(asBoth(asJson[HttpBinResponse], asStringAlways))

  val backend: SttpBackend[Identity, Any] = HttpURLConnectionBackend()

  try {
    val response: Response[(Either[ResponseException[String, circe.Error],␣
↪HttpBinResponse], String)] =
      request.send(backend)

    val (parsed, raw) = response.body

    println("Got response - parsed: " + parsed)
    println("Got response - raw: " + raw)

  } finally backend.close()
}
```

4.6 Model classes

sttp model is a stand-alone project which provides a basic HTTP model, along with constants for common HTTP header names, media types, and status codes.

The basic model classes are: `Header`, `Cookie`, `CookieWithMeta`, `MediaType`, `Method`, `StatusCode`, `CacheDirective`, `ETag` and `Uri`. The `.toString` methods of these classes returns a representation as in a HTTP request/response. See the ScalaDoc for more information.

Companion objects provide methods to construct model class instances, following these rules:

- `.parse(serialized: String): Either[String, ModelClass]`: returns an error message, or an instance of the model class
- `.unsafeParse(serialized: String): Sth`: returns an instance of the model class or in case of an error, throws an exception.
- `.unsafeApply(values)`: creates an instance of the model class; validates the input values and in case of an error, throws an exception. An error could be e.g. that the input values contain characters outside the allowed range
- `.safeApply(...): Either[String, ModelClass]`: same as above, but doesn't throw exceptions. Instead, returns an error message, or the model class instance
- `.apply(...): ModelClass`: creates the model type, without validation, and without throwing exceptions

Moreover, companion objects provide constants and/or constructor methods for well-know model class instances. For example, there's `StatusCode.Ok`, `Method.POST`, `MediaType.ImageGif` and `Header.contentType(MediaType)`.

These constants are also available as traits: `StatusCodes`, `MediaTypes` and `HeaderNames`.

The model also contains aggregate/helper classes such as `Headers` and `QueryParams`.

Example with objects:

```
import sttp.client3._
import sttp.model._

object Example {
  val request = basicRequest.header(Header.contentType(MediaType.ApplicationJson))
    .get(uri"https://httpbin.org")

  val backend = HttpURLConnectionBackend()
  val response = request.send(backend)
  if (response.code == StatusCode.Ok) println("Ok!")
}
```

Example with traits:

```
import sttp.client3._
import sttp.model._

object Example extends HeaderNames with MediaTypes with StatusCodes {
  val request = basicRequest.header(ContentType, ApplicationJson.toString)
    .get(uri"https://httpbin.org")

  val backend = HttpURLConnectionBackend()
  val response = request.send(backend)
```

(continues on next page)

(continued from previous page)

```
  if (response.code == Ok) println("Ok!")
}
```

For more information see

- Wikipedia: list of http header fields
- Wikipedia: media type
- Wikipedia: list of http status codes

4.7 URIs

A request can only be sent if the request method & URI are defined. To represent URIs, sttp comes with a `Uri` case class, which captures all the parts of an address.

To specify the request method and URI, use one of the methods on the request definition corresponding to the name of the desired HTTP method: `.post`, `.get`, `.put` etc. All of them accept a single parameter, the URI to which the request should be sent (these methods only modify the request definition; they don't send the requests).

The `Uri` class is immutable, and can be constructed by hand, but in many cases the URI interpolator will be easier to use.

4.7.1 URI interpolator

Using the URI interpolator it's possible to conveniently create `Uri` instances, for example:

```
import sttp.client3._
import sttp.model._

val user = "Mary Smith"
val filter = "programming languages"

val endpoint: Uri = uri"http://example.com/$user/skills?filter=$filter"

assert(endpoint.toString ==
  "http://example.com/Mary%20Smith/skills?filter=programming+languages")
```

Note the `uri` prefix before the string and the standard Scala string-embedding syntax (`$user`, `$filter`).

Any values embedded in the URI will be URL-encoded, taking into account the context (e.g., the whitespace in `user` will be %-encoded as `%20D`, while the whitespace in `filter` will be query-encoded as `+`). On the other hand, parts of the URI given as literal strings (not embedded values), are assumed to be URL-encoded and thus will be decoded when creating a `Uri` instance.

All components of the URI can be embedded from values: scheme, username/password, host, port, path, query and fragment. The embedded values won't be further parsed, except the `:` in the host part, which is commonly used to pass in both the host and port:

```
import sttp.client3._

// the embedded / is escaped
println(uri"http://example.org/${"a/b"}")
// http://example.org/a%2Fb
```

(continues on next page)

(continued from previous page)

```
// the embedded / is not escaped
println(uri"http://example.org/${"a"}/${"b"}")
// http://example.org/a/b

// the embedded : is not escaped
println(uri"http://${"example.org:8080"}")
// http://example.org:8080
```

Both the `Uri` class, and the interpolator can be used stand-alone, without using the rest of sttp. Conversions are available both from and to `java.net.URI`; `Uri.toString` returns the URI as a `String`.

4.7.2 Optional values

The URI interpolator supports optional values for hosts (subdomains), query parameters and the fragment. If the value is `None`, the appropriate URI component will be removed. For example:

```
val v1 = None
val v2 = Some("v2")
```

```
println(uri"http://example.com?p1=$v1&p2=v2")
// http://example.com?p2=v2

println(uri"http://$v1.$v2.example.com")
// http://v2.example.com

println(uri"http://example.com#$v1")
// http://example.com
```

4.7.3 Maps and sequences

Maps, sequences of tuples and sequences of values can be embedded in the query part. They will be expanded into query parameters. Maps and sequences of tuples can also contain optional values, for which mappings will be removed if `None`.

For example:

```
val ps = Map("p1" -> "v1", "p2" -> "v2")
```

```
println(uri"http://example.com?$ps&p3=p4")
// http://example.com?p1=v1&p2=v2&p3=p4
```

Sequences in the host part will be expanded to a subdomain sequence, and sequences in the path will be expanded to path components:

```
val params = List("a", "b", "c")
```

```
println(uri"http://example.com/$params")
// http://example.com/a/b/c
```

4.7.4 Special cases

If a string containing the protocol is embedded *at the very beginning*, it will not be escaped, allowing to embed entire addresses as prefixes, e.g.: `uri"$endpoint/login"`, where `val endpoint = "http://example.com/api"`.

This is useful when a base URI is stored in a value, and can then be used as a base for constructing more specific URIs.

4.7.5 Relative URIs

The `Uri` class can represent both relative and absolute URIs. Hence, in terms of rfc3986, it is in fact a URI reference.

Relative URIs can be created using the interpolator, same as absolute ones, e.g.:

```
println(uri"/api/$params")
// /api/a/b/c
```

When sending requests using relative URIs, the *ResolveRelativeUrisBackend* backend wrapper might be useful to resolve them.

4.7.6 All features combined

A fully-featured example:

```
import sttp.client3._
val secure = true
val scheme = if (secure) "https" else "http"
val subdomains = List("sub1", "sub2")
val vx = Some("y z")
val paramMap = Map("a" -> 1, "b" -> 2)
val jumpTo = Some("section2")
```

```
println(uri"$scheme://$subdomains.example.com?x=$vx&$paramMap#$jumpTo")
// https://sub1.sub2.example.com?x=y+z&a=1&b=2#section2
```

4.8 Request definition basics

As mentioned in the *quickstart*, the following import will be needed:

```
import sttp.client3._
```

This brings into scope `basicRequest`, the starting request. This request can be customised, each time yielding a new, immutable request definition (unless a mutable body is set on the request, such as a byte array). As the request definition is immutable, it can be freely stored in values, shared across threads, and customized multiple times in various ways.

For example, we can set a cookie, `String` -body and specify that this should be a `POST` request to a given URI:

```
val request = basicRequest
  .cookie("login", "me")
  .body("This is a test")
  .post(uri"http://endpoint.com/secret")
```

The request parameters (headers, cookies, body etc.) can be specified **in any order**. It doesn't matter if the request method, the body, the headers or connection options are specified in this sequence or another. This way you can build arbitrary request templates, capturing all that's common among your requests, and customizing as needed. Remember that each time a modifier is applied to a request, you get a new immutable object.

There's a lot of ways in which you can customize a request, which are covered in this guide. Another option is to just explore the API: most of the methods are self-explanatory and carry scaladocs if needed.

Using the modifiers, each time we get a new request definition, but it's just a description: a data object; nothing is sent over the network until the `send(backend)` method is invoked.

4.8.1 Query parameters and URIs

Query parameters are specified as part of the URI, to which the request should be sent. The URI can only be set together with the request method (using `.get(Uri)`, `.post(Uri)`, etc.).

The URI can be created programmatically (by calling methods on the `Uri` class), or using the `uri` interpolator, which also allows embedding (and later escaping) values from the environment. See the documentation on *creating URIs* for more details.

4.8.2 Sending a request

A request definition can be created without knowing how it will be sent. But to send a request, a backend is needed. A default, synchronous backend based on Java's `HttpURLConnection` is provided in the `core` jar.

To invoke the `send(backend)` method on a request description, you'll need an instance of `SttpBackend`:

```
val backend = HttpURLConnectionBackend()
val response: Identity[Response[Either[String, String]]] = request.send(backend)
```

The default backend uses the `Identity` effect to return responses, which is equivalent to a synchronous call (no effect at all). Other asynchronous backends use other effect types. See the section on *backends* for more details.

Note: Only requests with the request method and uri can be sent. If trying to send a request without these components specified, a compile-time error will be reported. On how this is implemented, see the documentation on the *type of request definitions*.

4.8.3 Initial requests

sttp provides two initial requests:

- `basicRequest`, which is an empty request with the `Accept-Encoding: gzip, deflate` header added. That's the one that is most commonly used.
- `emptyRequest`, a completely empty request, with no headers at all.

Both of these requests will by default read the response body into a UTF-8 `String`. How the response body is handled is also part of the request definition. See the section on *response body specifications* for more details on how to customize that.

4.8.4 Debugging requests

sttp comes with builtin request to curl converter. To convert request to curl invocation use `.toCurl` method.

For example:

```
basicRequest.get(uri"http://httpbin.org/ip").toCurl
// res1: String = "curl -L --max-redirs 32 -X GET 'http://httpbin.org/ip'"
```

Note that the `Accept-Encoding` header, which is added by default to all requests (`Accept-Encoding: gzip, deflate`) is filtered out from the generated command, so that when running a request from the command line, the result has higher chance of being human-readable, and not compressed.

4.9 Headers

Arbitrary headers can be set on the request using the `.header` method:

```
import sttp.client3._

basicRequest.header("User-Agent", "myapp")
```

As with any other request definition modifier, this method will yield a new request, which has the given header set. The headers can be set at any point when defining the request, arbitrarily interleaved with other modifiers.

While most headers should be set only once on a request, HTTP allows setting a header multiple times. That's why the `header` method has an additional optional boolean parameter, `replaceExisting`, which defaults to `true`. This way, if the same header is specified twice, only the last value will be included in the request. If previous values should be preserved, set this parameter to `false`.

There are also variants of this method accepting a number of headers:

```
import sttp.client3._
import sttp.model._

basicRequest.header(Header("k1", "v1"), replaceExisting = false)
basicRequest.header("k2", "v2")
basicRequest.header("k3", "v3", replaceExisting = true)
basicRequest.headers(Map("k4" -> "v4", "k5" -> "v5"))
basicRequest.headers(Header("k9", "v9"), Header("k10", "v10"), Header("k11", "v11"))
```

4.9.1 Common headers

For some common headers, dedicated methods are provided:

```
import sttp.client3._

basicRequest.contentType("application/json")
basicRequest.contentType("application/json", "iso-8859-1")
basicRequest.contentLength(128)
basicRequest.acceptEncoding("gzip, deflate")
```

See also documentation on setting cookies and authentication.

4.10 Cookies

Cookies sent in requests are key-value pairs contained in the `Cookie` header. They can be set on a request in a couple of ways. The first is using the `.cookie(name: String, value: String)` method. This will yield a new request definition which, when sent, will contain the given cookie.

Cookies are currently only available on the JVM.

Cookies can also be set using the following methods:

```
import sttp.client3._
import sttp.model.headers.CookieWithMeta

basicRequest
  .cookie("k1", "v1")
  .cookie("k2" -> "v2")
  .cookies("k3" -> "v3", "k4" -> "v4")
  .cookies(Seq(CookieWithMeta("k5", "k5"), CookieWithMeta("k6", "k6")))
```

4.10.1 Cookies from responses

It is often necessary to copy cookies from a response, e.g. after a login request is sent, and a successful response with the authentication cookie received. Having an object `response: Response[_]`, cookies on a request can be copied:

```
import sttp.client3._

val backend = HttpURLConnectionBackend()
val loginRequest = basicRequest
    .cookie("login", "me")
    .body("This is a test")
    .post(uri"http://endpoint.com")
val response = loginRequest.send(backend)

basicRequest.cookies(response)
```

Or, it's also possible to store only the `sttp.model.CookieWithMeta` objects (a sequence of which can be obtained from a response), and set the on the request:

```
import sttp.client3._

val backend = HttpURLConnectionBackend()
val loginRequest = basicRequest
    .cookie("login", "me")
    .body("This is a test")
    .post(uri"http://endpoint.com")
val response = loginRequest.send(backend)
val cookiesFromResponse = response.unsafeCookies

basicRequest.cookies(cookiesFromResponse)
```

4.11 Authentication

sttp supports basic, bearer-token based authentication and digest authentication. Two first cases are handled by adding an `Authorization` header with the appropriate credentials.

Basic authentication, using which the username and password are encoded using Base64, can be added as follows:

```
import sttp.client3._

val username = "mary"
val password = "p@assword"
basicRequest.auth.basic(username, password)
```

A bearer token can be added using:

```
val token = "zMDjRfl76ZC9Ub0wnz4XsNiRVBChTYbJcE3F"
basicRequest.auth.bearer(token)
```

4.11.1 Digest authentication

This type of authentication works differently. In its assumptions it is based on an additional message exchange between client and server. Due to that a special wrapping backend is need to handle that additional logic.

In order to add digest authentication support just wrap other backend as follows:

```
val myBackend: SttpBackend[Identity, Any] = HttpURLConnectionBackend()
new DigestAuthenticationBackend(myBackend)
```

Then only thing which we need to do is to pass our credentials to the relevant request:

```
val secureRequest = basicRequest.auth.digest(username, password)
```

It is also possible to use digest authentication against proxy:

```
val secureProxyRequest = basicRequest.proxyAuth.digest(username, password)
```

Both of above methods can be combined with different values if proxy and target server use digest authentication.

To learn more about digest authentication visit wikipedia

Also keep in mind that there are some limitations with the current implementation:

- there is no caching so each request will result in an additional round-trip (or two in case of proxy and server)
- authorizationInfo is not supported
- scalajs supports only md5 algorithm

4.11.2 OAuth2

You can use sttp with OAuth2. Looking at the OAuth2 protocol flow, sttp might be helpful in the second and third step of the process:

1. (A)/(B) - Your UI needs to enable the user to authenticate. Your application will then receive a callback from the authentication server, which will include an authentication code.

2. (C)/(D) - You need to send a request to the authentication server, passing in the authentication code from step 1. You'll receive an access token in response (and optionally a refresh token). For example, if you were using GitHub as your authentication server, you'd need to take the values of `clientId` and `clientSecret` from the GitHub settings, then take the `authCode` received in step 1 above, and send a request like this:

```
import sttp.client3.circe._
import io.circe._
import io.circe.generic.semiauto._

val authCode = "SplxlOBeZQQYbYS6WxSbIA"
val clientId = "myClient123"
val clientSecret = "s3cret"
case class MyTokenResponse(access_token: String, scope: String, token_type: String,␣
↪refresh_token: Option[String])
implicit val tokenResponseDecoder: Decoder[MyTokenResponse] =␣
↪deriveDecoder[MyTokenResponse]
val backend = HttpURLConnectionBackend()

val tokenRequest = basicRequest
    .post(uri"https://github.com/login/oauth/access_token?code=$authCode&grant_
↪type=authorization_code")
    .auth
    .basic(clientId, clientSecret)
    .header("accept","application/json")
val authResponse = tokenRequest.response(asJson[MyTokenResponse]).send(backend)
val accessToken = authResponse.body.map(_.access_token)
```

1. (E)/(F) - Once you have the access token, you can use it to request the protected resource from the resource server, depending on its specification.

4.12 Body

4.12.1 Text data

In its simplest form, the request's body can be set as a `String`. By default, this method will:

- use the UTF-8 encoding to convert the string to a byte array
- if not specified before, set `Content-Type: text/plain`
- if not specified before, set `Content-Length` to the number of bytes in the array

A `String` body can be set on a request as follows:

```
import sttp.client3._
basicRequest.body("Hello, world!")
```

It is also possible to use a different character encoding:

```
import sttp.client3._
basicRequest.body("Hello, world!", "utf-8")
```

4.12.2 Binary data

To set a binary-data body, the following methods are available:

```
import sttp.client3._

val bytes: Array[Byte] = ???
basicRequest.body(bytes)

import java.nio.ByteBuffer
val byteBuffer: ByteBuffer = ???
basicRequest.body(byteBuffer)

import java.io.ByteArrayInputStream
val inputStream: ByteArrayInputStream = ???
basicRequest.body(inputStream)
```

If not specified before, these methods will set the content type to `application/octet-stream`. When using a byte array, additionally the content length will be set to the length of the array (unless specified explicitly).

Note: While the object defining a request is immutable, setting a mutable request body will make the whole request definition mutable as well. With `InputStream`, the request can be moreover sent only once, as input streams can be consumed once.

4.12.3 Uploading files

To upload a file, simply set the request body as a `File` or `Path`:

```
import sttp.client3._

import java.io.File
basicRequest.body(new File("data.txt"))

import java.nio.file.Path
basicRequest.body(Path.of("data.txt"))
```

Note that on JavaScript only a `Web/API/File` is allowed.

As with binary body methods, the content type will default to `application/octet-stream`, and the content length will be set to the length of the file (unless specified explicitly).

See also multi-part and streaming support.

4.12.4 Form data

If you set the body as a `Map[String, String]` or `Seq[(String, String)]`, it will be encoded as form-data (as if a web form with the given values was submitted). The content type will default to `application/x-www-form-urlencoded`; content length will also be set if not specified.

By default, the `UTF-8` encoding is used, but can be also specified explicitly:

```
import sttp.client3._
basicRequest.body(Map("k1" -> "v1"))
basicRequest.body(Map("k1" -> "v1"), "utf-8")
basicRequest.body("k1" -> "v1", "k2" -> "v2")
basicRequest.body(Seq("k1" -> "v1", "k2" -> "v2"), "utf-8")
```

4.12.5 Custom body serializers

It is also possible to set custom types as request bodies, as long as there's an implicit `BodySerializer[B]` value in scope, which is simply an alias for a function:

```
type BodySerializer[B] = B => BasicRequestBody
```

A `BasicRequestBody` is a wrapper for one of the supported request body types: a `String`/byte array or an input stream.

For example, here's how to write a custom serializer for a case class, with serializer-specific default content type:

```
import sttp.client3._
import sttp.model.MediaType
case class Person(name: String, surname: String, age: Int)

// for this example, assuming names/surnames can't contain commas
implicit val personSerializer: BodySerializer[Person] = { p: Person =>
  val serialized = s"${p.name},${p.surname},${p.age}"
  StringBody(serialized, "UTF-8", MediaType.TextCsv)
}

basicRequest.body(Person("mary", "smith", 67))
```

See the implementations of the `BasicRequestBody` trait for more options.

4.13 Multipart requests

To set a multipart body on a request, the `multipartBody` method should be used (instead of `body`). Each body part is represented as an instance of `Part[BasicRequestBody]`, which can be conveniently constructed using `multipart` methods coming from the `sttp.client3` package.

A single part of a multipart request consist of a mandatory name and a payload of type:

- `String`
- `Array[Byte]`
- `ByteBuffer`
- `InputStream`
- `Map[String, String]`
- `Seq[(String, String)]`

To add a file part, the `multipartFile` method (also from the `com.softwaremill.sttp` package) should be used. This method is overloaded and supports `File`/`Path` objects on the JVM, and `Web/API/File` on JS.

The content type of each part is by default the same as when setting simple bodies: `text/plain` for parts of type `String`, `application/x-www-form-urlencoded` for parts of key-value pairs (form data) and `application/octet-stream` otherwise (for binary data).

The parts can be specified using either a `Seq[Multipart]` or by using multiple arguments:

```
import sttp.client3._

basicRequest.multipartBody(Seq(multipart("p1", "v1"), multipart("p2", "v2")))
basicRequest.multipartBody(multipart("p1", "v1"), multipart("p2", "v2"))
```

For example:

```
import sttp.client3._
import java.io._

val someFile = new File("/sample/path")

basicRequest.multipartBody(
  multipart("text_part", "data1"),
  multipartFile("file_part", someFile), // someFile: File
  multipart("form_part", Map("x" -> "10", "y" -> "yes"))
)
```

4.13.1 Customising part meta-data

For each part, an optional filename can be specified, as well as a custom content type and additional headers. For example:

```
import sttp.client3._
import java.io._

val logoFile = new File("/sample/path/logo123.jpg")
val docFile = new File("/sample/path/doc123.doc")
basicRequest.multipartBody(
  multipartFile("logo", logoFile).fileName("logo.jpg").contentType("image/jpg"),
  multipartFile("text", docFile).fileName("text.doc")
)
```

4.14 Streaming

Some backends (see *backends summary*) support streaming bodies, as described by the `Streams[S]` capability. If that's the case, you can set a stream of the supported type as a request body using the `streamBody` method, instead of the usual `body` method.

Note: Here, streaming refers to (usually) non-blocking, asynchronous streams of data. To send data which is available as an `InputStream`, or a file from local storage (which is available as a `File` or `Path`), no special backend support is needed. See the documenttation on *setting the request body*.

An implementation of the `Streams[S]` capability must be passed to the `.streamBody` method, to determine the type of streams that are supported. These implementations are provided by the backend implementations, e.g. `AkkaStreams` or `Fs2Streams[F]`.

For example, using the *akka-http backend*, a request with a streaming body can be defined as follows:

```
import sttp.client3._
import sttp.capabilities.akka.AkkaStreams

import akka.stream.scaladsl.Source
import akka.util.ByteString

val chunks = "Streaming test".getBytes("utf-8").grouped(10).to(Iterable)
val source: Source[ByteString, Any] = Source.apply(chunks.toList.map(ByteString(_)))
```

(continues on next page)

(continued from previous page)

```
basicRequest
  .streamBody(AkkaStreams)(source)
  .post(uri"...")
```

Note: A request with the body set as a stream can only be sent using a backend supporting exactly the given type of streams.

It's also possible to specify that the response body should be a stream.

4.15 The type of request definitions

All request definitions have type `RequestT[U, T, R]` (RequestT as in Request Template). If this looks a bit complex, don't worry, what the three type parameters stand for is the only thing you'll hopefully have to remember when using the API!

Going one-by-one:

- `U[_]` specifies if the request method and URL are specified. Using the API, this can be either `type Empty[X] = None`, meaning that the request has neither a method nor an URI. Or, it can be `type Id[X] = X` (type-level identity), meaning that the request has both a method and an URI specified. Only requests with a specified URI & method can be sent.
- `T` specifies the type to which the response will be read. By default, this is `Either[String, String]`. But it can also be e.g. `Array[Byte]` or `Unit`, if the response should be ignored. Response body handling can be changed by calling the `.response` method. With backends which support streaming, this can also be a supported stream type. See *response body specifications* for more details.
- `R` specifies the requirements of this request. Most of the time this will be `Any`, meaning that this request does not have any special requirements, and can be sent using any backend. So most of the time you can just ignore that parameter. However, if you are using streaming request/*response* bodies or *websockets*, the type parameter will reflect the required capabilities. They can include `Effect[F]`, `Streams[S]` and `WebSockets`.

There are two type aliases for the request template that are used:

- `type Request[T, R] = RequestT[Identity, T, R]`. A sendable request.
- `type PartialRequest[T, R] = RequestT[Empty, T, R]`

As `basicRequest`, the starting request, by default reads the body into a `Either[String, String]`, its type is:

```
basicRequest:  PartialRequest[Either[String, String], Any]
```

4.16 Responses

Responses are represented as instances of the case class `Response[T]`, where `T` is the type of the response body. When sending a request, an effect containing the response will be returned. For example, for asynchronous backends, we can get a `Future[Response[T]]`, while for the default synchronous backend, the wrapper will be a no-op, `Identity`, which is the same as no wrapper at all.

If sending the request fails, either due to client or connection errors, an exception will be thrown (synchronous backends), or a failed effect will be returned (e.g. a failed future).

Note: If the request completes, but results in a non-2xx return code, the request is still considered successful, that is, a `Response[T]` will be returned. See *response body specifications* for details on how such cases are handled.

4.16.1 Response code

The response code is available through the `.code` property. There are also methods such as `.isSuccess` or `.isServerError` for checking specific response code ranges.

4.16.2 Response headers

Response headers are available through the `.headers` property, which gives all headers as a sequence (not as a map, as there can be multiple headers with the same name).

Individual headers can be obtained using the methods:

```
import sttp.model._
import sttp.client3._
val backend = HttpURLConnectionBackend()
val request = basicRequest
    .get(uri"http://endpoint.com/example")
val response = request.send(backend)

val singleHeader: Option[String] = response.header(HeaderNames.Server)
val multipleHeaders: Seq[String] = response.headers(HeaderNames.Allow)
```

There are also helper methods available to read some commonly accessed headers:

```
val contentType: Option[String] = response.contentType
val contentLength: Option[Long] = response.contentLength
```

Finally, it's possible to parse the response cookies into a sequence of the `CookieWithMeta` case class:

```
import sttp.model.headers.CookieWithMeta

val cookies: Seq[CookieWithMeta] = response.unsafeCookies
```

If the cookies from a response should be set without changes on the request, this can be done directly; see the *cookies* section in the request definition documentation.

4.16.3 Obtaining the response body

The response body can be obtained through the `.body: T` property. `T` is the body deserialized as specified in the request description - see the next section on response body specifications.

4.17 Response body specification

By default, the received response body will be read as a `Either[String, String]`, using the encoding specified in the `Content-Type` response header (and if none is specified, using `UTF-8`). This is of course configurable: response bodies can be ignored, deserialized into custom types, received as a stream or saved to a file.

The default `response.body` will be a:

- `Left(errorMessage)` if the request is successful, but response code is not 2xx.
- `Right(body)` if the request is successful, and the response code is 2xx.

How the response body will be read is part of the request description, as already when sending the request, the backend needs to know what to do with the response. The type to which the response body should be deserialized is the second type parameter of `RequestT`, and stored in the request definition as the `request.response: ResponseAs[T, R]` property.

4.17.1 Basic response specifications

To conveniently specify how to deserialize the response body, a number of `as(...Type...)` methods are available. They can be used to provide a value for the request description's `response` property:

```
import sttp.client3._

basicRequest.response(asByteArray)
```

When the above request is completely described and sent, it will result in a `Response[Either[String, Array[Byte]]]` (where the left and right correspond to non-2xx and 2xx status codes, as above).

Other possible response descriptions include (the first type parameter of `ResponseAs` specifies the type returned as the response body, the second - the capabilities that the backend is required to support to send the request; `Any` means no special requirements):

```
import sttp.client3._
import java.io.File
import java.nio.file.Path

def ignore: ResponseAs[Unit, Any] = ???
def asString: ResponseAs[Either[String, String], Any] = ???
def asStringAlways: ResponseAs[String, Any] = ???
def asString(encoding: String): ResponseAs[Either[String, String], Any] = ???
def asStringAlways(encoding: String): ResponseAs[String, Any] = ???
def asByteArray: ResponseAs[Either[String, Array[Byte]], Any] = ???
def asByteArrayAlways: ResponseAs[Array[Byte], Any] = ???
def asParams: ResponseAs[Either[String, Seq[(String, String)]], Any] = ???
def asParamsAlways: ResponseAs[Seq[(String, String)], Any] = ???
def asParams(encoding: String): ResponseAs[Either[String, Seq[(String, String)]],␣
↪Any] = ???
def asParamsAlways(encoding: String): ResponseAs[Seq[(String, String)], Any] = ???
def asFile(file: File): ResponseAs[Either[String, File], Any] = ???
def asFileAlways(file: File): ResponseAs[File, Any] = ???
def asPath(path: Path): ResponseAs[Either[String, Path], Any] = ???
def asPathAlways(path: Path): ResponseAs[Path, Any] = ???

def asEither[A, B, R](onError: ResponseAs[A, R],
                      onSuccess: ResponseAs[B, R]): ResponseAs[Either[A, B], R] = ???
def fromMetadata[T, R](default: ResponseAs[T, R],
                       conditions: ConditionalResponseAs[T, R]*): ResponseAs[T, R] = ?
↪??

def asBoth[A, B](l: ResponseAs[A, Any], r: ResponseAs[B, Any]): ResponseAs[(A, B),␣
↪Any] = ???
def asBothOption[A, B, R](l: ResponseAs[A, R], r: ResponseAs[B, Any]): ResponseAs[(A,␣
↪Option[B]), R] = ???
```

Hence, to discard the response body, the request description should include the following:

```
import sttp.client3._

basicRequest.response(ignore)
```

And to save the response to a file:

```
import sttp.client3._
import java.io._

val someFile = new File("some/path")
basicRequest.response(asFile(someFile))
```

Note: As the handling of response is specified upfront, there's no need to "consume" the response body. It can be safely discarded if not needed.

4.17.2 Failing when the response code is not 2xx

Sometimes it's convenient to get a failed effect (or an exception thrown) when the response status code is not successful. In such cases, the response specification can be modified using the `.getRight` combinator:

```
import sttp.client3._

basicRequest.response(asString.getRight): PartialRequest[String, Any]
```

The combinator works in all cases where the response body is specified to be deserialized as an `Either`. If the left is already an exception, it will be thrown unchanged. Otherwise, the left-value will be wrapped in an `HttpError`.

Note: While both `asStringAlways` and `asString.getRight` have the type `ResponseAs[String, Any]`, they are different. The first will return the response body as a string always, regardless of the responses' status code. The second will return a failed effect / throw a `HttpError` exception for non-2xx status codes, and the string as body only for 2xx status codes.

There's also a variant of the combinator, `.getEither`, which can be used to extract typed errors and fail the effect if there's a deserialization error.

4.17.3 Custom body deserializers

It's possible to define custom body deserializers by taking any of the built-in response descriptions and mapping over them. Each `ResponseAs` instance has `map` and `mapWithMetadata` methods, which can be used to transform it to a description for another type (optionally using response metadata, such as headers or the status code). Each such value is immutable and can be used multiple times.

Note: Alternatively, response descriptions can be modified directly from the request description, by using the `request.mapResponse(...)` and `request.mapResponseRight(...)` methods (which is available, if the response body is deserialized to an either). That's equivalent to calling `request.response(request.response.map(...))`, that is setting a new response description, to a modified old response description; but with shorter syntax.

As an example, to read the response body as an int, the following response description can be defined (warning: this ignores the possibility of exceptions!):

```
import sttp.client3._

val asInt: ResponseAs[Either[String, Int], Any] = asString.mapRight(_.toInt)

basicRequest
  .get(uri"http://example.com")
  .response(asInt)
```

To integrate with a third-party JSON library, and always parse the response as a json (regardless of the status code):

```
import sttp.client3._

type JsonError
type JsonAST

def parseJson(json: String): Either[JsonError, JsonAST] = ???
val asJson: ResponseAs[Either[JsonError, JsonAST], Any] = asStringAlways.
↪map(parseJson)

basicRequest
  .response(asJson)
```

A number of JSON libraries are supported out-of-the-box, see *json support*.

4.17.4 Response-metadata dependent deserializers

Using the `fromMetadata` combinator, it's possible to dynamically specify how the response should be deserialized, basing on the response status code and response headers. The default `asString`, `asByteArray` response descriptions use this method to return a `Left` in case of non-2xx responses, and a `Right` otherwise.

A more complex case, which uses Circe for deserializing JSON, choosing to which model to deserialize to depending on the status code, can look as following:

```
import sttp.client3._
import sttp.model._
import sttp.client3.circe._
import io.circe._
import io.circe.generic.auto._

sealed trait MyModel
case class SuccessModel(name: String, age: Int) extends MyModel
case class ErrorModel(message: String) extends MyModel

val myRequest: Request[Either[ResponseException[String, io.circe.Error], MyModel],␣
↪Nothing] =
  basicRequest
    .get(uri"https://example.com")
    .response(fromMetadata(
        asJson[ErrorModel],
        ConditionalResponseAs(_.code == StatusCode.Ok, asJson[SuccessModel])
    ))
```

The above example assumes that success and error models are part of one hierarchy (`MyModel`). Sometimes http errors are modelled independently of success. In this case, we can use `asJsonEither`, which uses

`asEitherDeserialized` under the covers:

```
import sttp.client3._
import sttp.model._
import sttp.client3.circe._
import io.circe._
import io.circe.generic.auto._

case class MyModel(p1: Int)
sealed trait MyErrorModel
case class Conflict(message: String) extends MyErrorModel
case class BadRequest(message: String) extends MyErrorModel
case class GenericError(message: String) extends MyErrorModel

basicRequest
  .get(uri"https://example.com")
  .response(asJsonEither[MyErrorModel, MyModel])
```

4.17.5 Streaming

If the backend used supports streaming (see *backends summary*), it's possible to receive responses as a stream. This can be described using the following methods:

```
import sttp.capabilities.{Effect, Streams}
import sttp.client3._
import sttp.model.ResponseMetadata

def asStream[F[_], T, S](s: Streams[S])(f: s.BinaryStream => F[T]):
  ResponseAs[Either[String, T], Effect[F] with S] = ???

def asStreamWithMetadata[F[_], T, S](s: Streams[S])(
      f: (s.BinaryStream, ResponseMetadata) => F[T]
  ): ResponseAs[Either[String, T], Effect[F] with S] = ???

def asStreamAlways[F[_], T, S](s: Streams[S])(f: s.BinaryStream => F[T]):
  ResponseAs[T, Effect[F] with S] = ???

def asStreamAlwaysWithMetadata[F[_], T, S](s: Streams[S])(
      f: (s.BinaryStream, ResponseMetadata) => F[T]
  ): ResponseAs[T, Effect[F] with S] = ???

def asStreamUnsafe[S](s: Streams[S]):
  ResponseAs[Either[String, s.BinaryStream], S] = ???

def asStreamUnsafeAlways[S](s: Streams[S]):
  ResponseAs[s.BinaryStream, S] = ???
```

All of these specifications require the streaming capability to be passes as a parameter, an implementation of `Streams[S]`. This is used to determine the type of binary streams that are supported, and to require that the backend used to send the request supports the given type of streams. These implementations are provided by the backend implementations, e.g. `AkkaStreams` or `Fs2Streams[F]`.

The first two "safe" variants passes the response stream to the user-provided function, which should consume the stream entirely. Once the effect returned by the function is complete, the backend will try to close the stream (if the streaming implementation allows it).

The "unsafe" variants return the stream directly to the user, and then it's up to the user of the code to consume and

close the stream, releasing any resources held by the HTTP connection.

For example, when using the *Akka backend*:

```
import akka.stream.scaladsl.Source
import akka.util.ByteString
import scala.concurrent.Future
import sttp.capabilities.akka.AkkaStreams
import sttp.client3._
import sttp.client3.akkahttp.AkkaHttpBackend

val backend: SttpBackend[Future, AkkaStreams] = AkkaHttpBackend()

val response: Future[Response[Either[String, Source[ByteString, Any]]]] =
  basicRequest
    .post(uri"...")
    .response(asStreamUnsafe(AkkaStreams))
    .send(backend)
```

It's also possible to parse the received stream as server-sent events (SSE), using an implementation-specific mapping function. Refer to the documentation for particular backends for more details.

4.18 Exceptions

HTTP requests might fail in a variety of ways! There are two basic types of failures that might occur:

- network-level failure, such as the invalid/unroutable hosts, inability to establish a TCP connection, or broken sockets
- protocol-level failure, represented as 4xx and 5xx responses

The first type of failures is represented by exceptions, which are thrown when sending the request (using `request.send(backend)`). The second type of failure is represented as a `Response[T]`, with the appropriate response code. The response body might depend on the status code; by default the response is read as a `Either[String, String]`, where the left side represents protocol-level failure, and the right side: success.

Exceptions might be thrown directly (`Identity` synchronous backends), or returned as failed effects (other backends, e.g. failed `scala.concurrent.Future`). Backends will try to categorise these exceptions into a `SttpClientException`, which has two subclasses:

- `ConnectException`: when a connection (tcp socket) can't be established to the target host
- `ReadException`: when a connection has been established, but there's any kind of problem receiving the response (e.g. a broken socket)

In general, it's safe to assume that the request hasn't been sent in case of connect exceptions. With read exceptions, the target host might or might have not received and processed the request.

Unknown exceptions aren't categorised and are re-thrown unchanged.

4.18.1 Deserialization errors

Exceptions might also be thrown when deserializing the response body - depending on the specification of how to handle response bodies. The built-in deserializers (see e.g. *json*) return errors represented as `ResponseException[HE, DE]`, which can either be a `HttpError` (protocol-level failures, containing a potentially deserialized body value) or a `DeserializationException` (containing a deserialization-library-specific exception).

This means that a typical `asJson` response specification will result in the body being read as:

```
import sttp.client3._
def asJson[T]: ResponseAs[Either[ResponseException[String, Exception], T], Any] = ???
```

There are also the `.getRight` and `.getEither` methods on eligible response specifications, which convert http errors or deserialization exceptions as failed effects.

4.18.2 Possible outcomes

Summing up, when the response is deserialized (e.g. to json), sending a request can have these outcomes, and can be represented in the following ways:

- network-level success (HTTP request sent and response received)
  - http error (4xx, 5xx), successfully parsed (**a value wrapped in `Left`, or a failed effect**)
  - http success (2xx), successfully parsed (**a value possibly wrapped in `Right`**)
  - deseralization error (**a value wrapped in `Left`, or a failed effect**)
- network-level failure (invalid host, broken socket): failed effect

4.19 WebSockets

One of the optional capabilities (represented as `WebSockets`) that a backend can support are websockets (see *backends summary*). Websocket requests are described exactly the same as regular requests, starting with `basicRequest`, adding headers, specifying the request method and uri.

A websocket request will be sent instead of a regular one if the response specification includes handling the response as a websocket. A number of `asWebSocket(...)` methods are available, giving a couple of variants of working with websockets.

4.19.1 Using `WebSocket`

The first possibility is using `sttp.client3.ws.WebSocket[F]`, where `F` is the backend-specific effects wrapper, such as `Future` or `IO`. It contains two basic methods, both of which use the `F` effect to return results:

- `def receive: F[WebSocketFrame]` which will complete once a message is available, and return the next incoming frame (which can be a data, ping, pong or close)
- `def send(f: WebSocketFrame, isContinuation: Boolean = false): F[Unit]`, which sends a message to the websocket. The `WebSocketFrame` companion object contains methods for creating binary/text messages. When using fragmentation, the first message should be sent using `finalFragment = false`, and subsequent messages using `isContinuation = true`.

The `WebSocket` trait also contains other methods for receiving only text/binary messages, as well as automatically sending `Pong` responses when a `Ping` is received.

The following response specifications which use `WebSocket[F]` are available (the first type parameter of `ResponseAs` specifies the type returned as the response body, the second - the capabilities that the backend is required to support to send the request):

```
import sttp.client3._
import sttp.capabilities.{Effect, WebSockets}
import sttp.model.ResponseMetadata
import sttp.ws.WebSocket

def asWebSocket[F[_], T](f: WebSocket[F] => F[T]):
  ResponseAs[Either[String, T], Effect[F] with WebSockets] = ???

def asWebSocketWithMetadata[F[_], T](
      f: (WebSocket[F], ResponseMetadata) => F[T]
  ): ResponseAs[Either[String, T], Effect[F] with WebSockets] = ???

def asWebSocketAlways[F[_], T](f: WebSocket[F] => F[T]):
  ResponseAs[T, Effect[F] with WebSockets] = ???

def asWebSocketAlwaysWithMetadata[F[_], T](
      f: (WebSocket[F], ResponseMetadata) => F[T]
  ): ResponseAs[T, Effect[F] with WebSockets] = ???

def asWebSocketUnsafe[F[_]]:
  ResponseAs[Either[String, WebSocket[F]], Effect[F] with WebSockets] = ???

def asWebSocketUnsafeAlways[F[_]]:
  ResponseAs[WebSocket[F], Effect[F] with WebSockets] = ???
```

The first variant, `asWebSocket`, passes an open `WebSocket` to the user-provided function. This function should return an effect which completes, once interaction with the websocket is finished. The backend can then safely close the websocket. The value that's returned as the response body is either an error (represented as a `String`), in case the websocket upgrade didn't complete successfully, or the value returned by the websocket-interacting method.

The second variant (`asWebSocketAlways`) is similar, but any errors due to failed websocket protocol upgrades are represented as failed effects (exceptions).

The remaining two variants return the open `WebSocket` directly, as the response body. It is then the responsibility of the client code to close the websocket, once it's no longer needed.

See also the *examples*, which include examples involving websockets.

4.19.2 Using streams

Another possibility is to work with websockets by providing a streaming stage, which transforms incoming data frames into outgoing frames. This can be e.g. an *Akka* `Flow` or a *fs2* `Pipe`.

The following response specifications are available:

```
import sttp.client3._
import sttp.capabilities.{Streams, WebSockets}
import sttp.ws.WebSocketFrame

def asWebSocketStream[S](s: Streams[S])(p: s.Pipe[WebSocketFrame.Data[_],␣
↪WebSocketFrame]):
  ResponseAs[Either[String, Unit], S with WebSockets] = ???

def asWebSocketStreamAlways[S](s: Streams[S])(p: s.Pipe[WebSocketFrame.Data[_],␣
↪WebSocketFrame]):
  ResponseAs[Unit, S with WebSockets] = ???
```

Using streaming websockets requires the backend to support the given streaming capability (see also *streaming*). Streaming capabilities are described as implementations of `Streams[S]`, and are provided by backend implementations, e.g. `AkkaStreams` or `Fs2Streams[F]`.

When working with streams of websocket frames keep in mind that a text payload maybe fragmented into multiple frames. sttp provides two useful methods (`fromTextPipe`, `fromTextPipeF`) for each backend to aggregate these fragments back into complete messages. These methods can be found in corresponding WebSockets classes for given effect type:

effect type	class name
`monix.Task`	`sttp.client3.impl.monix.MonixWebSockets`
`ZIO`	`sttp.client3.impl.zio.ZioWebSockets`
`fs2.Stream`	`sttp.client3.impl.fs2.Fs2WebSockets`

4.20 JSON

Adding support for JSON (or other format) bodies in requests/responses is a matter of providing a *body serializer* and/or a *response body specification*. Both are quite straightforward to implement, so integrating with your favorite JSON library shouldn't be a problem. However, there are some integrations available out-of-the-box.

Each integration is available as an import, which brings the implicit `BodySerializers` and `asJson` methods into scope. Alternatively, these values are grouped intro traits (e.g. `sttp.client3.circe.SttpCirceApi`), which can be extended to group multiple integrations in one object, and thus reduce the number of necessary imports.

The following variants of `asJson` methods are available:

- regular - deserializes the body to json, only if the response is successful (2xx)
- `always` - deserializes the body to json regardless of the status code
- `either` - uses different deserializers for error and successful (2xx) responses

The type signatures vary depending on the underlying library (required implicits and error representation differs), but they obey the following pattern:

```
import sttp.client3._

def asJson[B]: ResponseAs[Either[ResponseException[String, Exception], B], Any] = ???
def asJsonAlways[B]: ResponseAs[Either[DeserializationException[Exception], B], Any]␣
↪= ???
def asJsonEither[E, B]: ResponseAs[Either[ResponseException[E, Exception], B], Any] =␣
↪???
```

The response specifications can be further refined using `.getRight` and `.getEither`, see *response body specifications*.

Following data class will be used through the next few examples:

```
case class RequestPayload(data: String)
case class ResponsePayload(data: String)
```

4.20.1 Circe

JSON encoding of bodies and decoding of responses can be handled using Circe by the `circe` module. To use add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "circe" % "3.1.3"
```

This module adds a body serialized, so that json payloads can be sent as request bodies. To send a payload of type `T` as json, a `io.circe.Encoder[T]` implicit value must be available in scope. Automatic and semi-automatic derivation of encoders is possible by using the circe-generic module.

Response can be parsed into json using `asJson[T]`, provided there's an implicit `io.circe.Decoder[T]` in scope. The decoding result will be represented as either a http/deserialization error, or the parsed value. For example:

```
import sttp.client3._
import sttp.client3.circe._

val backend: SttpBackend[Identity, Any] = HttpURLConnectionBackend()

import io.circe.generic.auto._
val requestPayload = RequestPayload("some data")

val response: Identity[Response[Either[ResponseException[String, io.circe.Error],␣
↪ResponsePayload]]] =
  basicRequest
    .post(uri"...")
    .body(requestPayload)
    .response(asJson[ResponsePayload])
    .send(backend)
```

Arbitrary JSON structures can be traversed by parsing the result as `io.circe.Json`, and using the circe-optics module.

4.20.2 Json4s

To encode and decode json using json4s, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "json4s" % "3.1.3"
"org.json4s" %% "json4s-native" % "3.6.0"
```

Note that in this example we are using the json4s-native backend, but you can use any other json4s backend.

Using this module it is possible to set request bodies and read response bodies as case classes, using the implicitly available `org.json4s.Formats` (which defaults to `org.json4s.DefaultFormats`), and by bringing an implicit `org.json4s.Serialization` into scope.

Usage example:

```
import sttp.client3._
import sttp.client3.json4s._

val backend: SttpBackend[Identity, Any] = HttpURLConnectionBackend()

val requestPayload = RequestPayload("some data")

implicit val serialization = org.json4s.native.Serialization
implicit val formats = org.json4s.DefaultFormats

val response: Identity[Response[Either[ResponseException[String, Exception],␣
↪ResponsePayload]]] =
  basicRequest
```

(continues on next page)

(continued from previous page)

```
    .post(uri"...")
    .body(requestPayload)
    .response(asJson[ResponsePayload])
    .send(backend)
```

4.20.3 spray-json

To encode and decode JSON using spray-json, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "spray-json" % "3.1.3"
```

Using this module it is possible to set request bodies and read response bodies as your custom types, using the implicitly available instances of `spray.json.JsonWriter` / `spray.json.JsonReader` or `spray.json.JsonFormat`.

Usage example:

```
import sttp.client3._
import sttp.client3.sprayJson._
import spray.json._

val backend: SttpBackend[Identity, Any] = HttpURLConnectionBackend()

implicit val payloadJsonFormat: RootJsonFormat[RequestPayload] = ???
implicit val myResponseJsonFormat: RootJsonFormat[ResponsePayload] = ???

val requestPayload = RequestPayload("some data")

val response: Identity[Response[Either[ResponseException[String, Exception],␣
↪ResponsePayload]]] =
  basicRequest
    .post(uri"...")
    .body(requestPayload)
    .response(asJson[ResponsePayload])
    .send(backend)
```

4.20.4 play-json

To encode and decode JSON using play-json, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "play-json" % "3.1.3"
```

To use, add an import: `import sttp.client3.playJson._`.

4.21 Resilience

Resilience covers areas such as retries, circuit breaking and rate limiting.

sttp client doesn't have the above built-in, as these concepts are usually best handled on a higher level. Sending a request (that is, invoking `myRequest.send(backend)`), can be viewed as a:

- `() => Response[T]` function for synchronous backends

- `() => Future[Response[T]]` for `Future`-based asynchronous backends
- `IO[Response[T]]`/`Task[Response[T]]` process description

All of these are lazily evaluated, and can be repeated. Such a representation allows to integrate the `send()` side-effect with a stack-dependent resilience tool. There's a number of libraries that implement the above mentioned resilience functionalities, hence there's no sense for sttp client to reimplement any of those. That's simply not the scope of this library.

Still, the input for a particular resilience model might involve both the result (either an exception, or a response) and the original description of the request being sent. E.g. retries can depend on the request method; circuit-breaking can depend on the host, to which the request is sent; same for rate limiting.

4.21.1 Retries

Here's an incomplete list of libraries which can be used to manage retries in various Scala stacks:

- for `Future`: retry
- for ZIO: schedules, rezilience
- for Monix/cats-effect: cats-retry
- for Monix: `.restart` methods

sttp client contains a default implementation of a predicate, which allows deciding if a request is retriable: if the body can be sent multiple times, and if the HTTP method is idempotent. This predicate is available as `RetryWhen.Default` and has type `(Request[_, _], Either[Throwable, Response[_]]) => Boolean`.

See also the retrying using ZIO example, as well as an example of a very simple retrying backend wrapper.

Note that some backends also have built-in retry mechanisms, e.g. akka-http or OkHttp (see the builder's `retryOnConnectionFailure` method).

4.21.2 Circuit breaking

- for Monix & cats-effect: monix-catnap
- for Akka/`Future`: akka circuit breaker
- for ZIO: rezilience

4.21.3 Rate limiting

- for akka-streams: throttle in akka streams
- for ZIO: rezilience

4.21.4 Java libraries

- resilience4j

4.22 OpenAPI

sttp-client *request definitions* can be automatically generated from openapi `.yaml` specifications using the `scala-sttp` code generator, included in the openapi-generator project.

For `scala-sttp`'s generator's configuration options refer to: https://openapi-generator.tech/docs/generators/scala-sttp.

4.22.1 Standalone setup

This is the simplest setup which relies on calling openapi-generator manually and generating a complete sbt project from it.

First, you will need to install/download openapi-generator. Follow openapi-generator's official documentation on how to do this.

Keep in mind that the `scala-sttp` generator is available only since v5.0.0-beta.

Next, call the generator with the following options:

```
openapi-generator-cli generate \
  -i petstore.yaml \
  --generator-name scala-sttp \
  -o samples/client/petstore/
```

4.22.2 Sbt managed

In this setup openapi-generator is plugged into sbt project through the sbt-openapi-generator plugin. Sttp requests and models are automatically generated upon compilation.

To have your openapi descriptions automatically turned into classes first define a new module in your project:

```
lazy val petstoreApi: Project = project
  .in(file("petstore-api"))
  .settings(
    openApiInputSpec := s"${baseDirectory.value.getPath}/petstore.yaml",
    openApiGeneratorName := "scala-sttp",
    openApiOutputDir := baseDirectory.value.name,
    libraryDependencies ++= Seq(
      "com.softwaremill.sttp.client3" %% "core" % "3.1.3",
      "com.softwaremill.sttp.client3" %% "json4s" % "3.1.3",
      "org.json4s" %% "json4s-jackson" % "3.6.8"
    )
  )
```

As this will generate code into `petstore-api/src` you might want to add this folder to the `.gitignore`.

Since this plugin is still in a very early stage it requires some additional configuration.

First we need to connect generation with compilation. Add following line into petstore module settings:

```
    (compile in Compile) := ((compile in Compile) dependsOn openApiGenerate).value,
```

Now we have to attach our generated source code directory into cleaning process. Add following line into petstore module settings:

```
    cleanFiles += baseDirectory.value / "src"
```

Last but not least we need to tell openapi-generator not to generate whole project but only the source files (without the sbt build file): Add following line into petstore module settings:

```
    openApiIgnoreFileOverride := s"${baseDirectory.in(ThisBuild).value.getPath}/
↪openapi-ignore-file",
```

and create `openapi-ignore-file` file in project's root directory with following content:

```
*
**/*
!**/src/main/scala/**/*
```

Final petstore module configuration:

```
lazy val petstoreApi: Project = project
  .in(file("petstore-api"))
  .settings(
    openApiInputSpec := s"${baseDirectory.value.getPath}/petstore.yaml",
    openApiGeneratorName := "scala-sttp",
    openApiOutputDir := baseDirectory.value.name,
    openApiIgnoreFileOverride := s"${baseDirectory.in(ThisBuild).value.getPath}/
↪openapi-ignore-file",
    libraryDependencies ++= Seq(
      "com.softwaremill.sttp.client3" %% "core" % "3.1.3",
      "com.softwaremill.sttp.client3" %% "json4s" % "3.1.3",
      "org.json4s" %% "json4s-jackson" % "3.6.8"
    ),
    (compile in Compile) := ((compile in Compile) dependsOn openApiGenerate).value,
    cleanFiles += baseDirectory.value / "src"
  )
```

Full demo project is available on github.

Additional notes

Although recent versions of the IntelliJ IDEA IDE come with "OpenApi Specification" plugin bundled into it, this plugin doesn't seem to support latest versions of generator and so, it is impossible to generate sttp bindings from it.

4.23 Supported backends

sttp supports a number of synchronous and asynchronous backends. It's the backends that take care of managing connections, sending requests and receiving responses: sttp defines only the API to describe the requests to be send and handle the response data. Backends do all the heavy-lifting.

Choosing the right backend depends on a number of factors: whether you are using sttp to explore some data, or is it a production system; are you using a synchronous, blocking architecture or an asynchronous one; do you work mostly with Scala's `Future`, or maybe you use some form of a `Task` abstraction; finally, if you want to stream requests/responses, or not.

Which one to choose?

- for simple exploratory requests, use the synchronous `HttpURLConnectionBackend`, or `HttpClientSyncBackend` if you are on Java11+.

- if you have Akka in your stack, use the Akka backend
- otherwise, if you are using `Future`, use `AsyncHttpClientFutureBackend` Future backend, or `HttpClientFutureBackend` if you are on Java11+.
- finally, if you are using a functional effect wrapper, use one of the "functional" backends, for ZIO, Monix, Scalaz, cats-effect or fs2.

Each backend has two type parameters:

- `F[_]`, the effects wrapper for responses. That is, when you invoke `send(backend)` on a request description, do you get a `Response[_]` directly, or is it wrapped in a `Future` or a `Task`?
- `P`, the capabilities supported by the backend, in addition to `Effect[F]`. If `Any`, no additional capabilities are provided. Might include `Streams` (the ability to send and receive streaming bodies) and `WebSockets` (the ability to handle websocket requests).

Below is a summary of all the JVM backends; see the sections on individual backend implementations for more information:

Class	Effect type	Supported stream type	Supports websockets	Fully non-blocking
HttpURLConnectionBackend	None (Identity)	n/a	no	no
TryHttpURLConnectionBackend	scala.util.Try	n/a	no	no
AkkaHttpBackend	scala.concurrent.Future	akka.stream.scaladsl.Source[ByteString, Any]	yes (regular & streaming)	yes
AsyncHttpClientFutureBackend	scala.concurrent.Future	n/a	yes (regular)	no
AsyncHttpClientScalazBackend	scalaz.concurrent.Task	n/a	yes (regular)	no
AsyncHttpClientZioBackend	zio.Task	zio.stream.Stream[Throwable, Byte]	yes (regular & streaming)	no
AsyncHttpClientMonixBackend	monix.eval.Task	monix.reactive.Observable[ByteBuffer]	yes (regular & streaming)	no
AsyncHttpClientCatsBackend	F[_]: cats.effect.Concurrent	n/a	no	no
AsyncHttpClientFs2Backend	F[_]: cats.effect.Concurrent	fs2.Stream[F, Byte]	yes (regular & streaming)	no
OkHttpSyncBackend	None (Identity)	n/a	yes (regular)	no
OkHttpFutureBackend	scala.concurrent.Future	n/a	yes (regular)	no
OkHttpMonixBackend	monix.eval.Task	monix.reactive.Observable[ByteBuffer]	yes (regular & streaming)	no
Http4sBackend	F[_]: cats.effect.Effect	fs2.Stream[F, Byte]	no	no
HttpClientSyncBackend	None (Identity)	n/a	no	no
HttpClientFutureBackend	scala.concurrent.Future	n/a	yes (regular)	no
HttpClientMonixBackend	monix.eval.Task	monix.reactive.Observable[ByteBuffer]	yes (regular & streaming)	yes
HttpClientFs2Backend	F[_]: cats.effect.Concurrent	fs2.Stream[F, Byte]	yes (regular & streaming)	yes
HttpClientZioBackend	zio.Task	zio.stream.Stream[Throwable, Byte]	yes (regular & streaming)	yes
FinagleBackend	com.twitter.util.Future	n/a	no	no

The backends work with Scala 2.11, 2.12 and 2.13 (with some exceptions for 2.11). Moreover, HttpURLConnectionBackend, AsyncHttpClientFutureBackend, AsyncHttpClientZioBackend, HttpClientSyncBackend, HttpClientFutureBackend and HttpClientZioBackend are additionally built with Scala 3.

All backends that support asynchronous/non-blocking streams, also support server-sent events.

There are also backends which wrap other backends to provide additional functionality. These include:

- `TryBackend`, which safely wraps any exceptions thrown by a synchronous backend in `scala.util.Try`
- `OpenTracingBackend`, for OpenTracing-compatible distributed tracing. See the dedicated section.
- `PrometheusBackend`, for gathering Prometheus-format metrics. See the dedicated section.
- extendable logging backends (with an slf4j implementation) backends. See the dedicated section.
- `ResolveRelativeUrisBackend` to resolve relative URIs given a base URI, or an arbitrary effectful function
- `ListenerBackend` to listen for backend lifecycle events. See the dedicated section.
- `FollowRedirectsBackend`, which handles redirects. All implementation backends are created wrapped with this one.

In addition, there are also backends for Scala.JS:

Class	Effect type	Supported stream type	Supports websockets
`FetchBackend`	`scala.concurrent.Future`	n/a	no
`FetchMonixBackend`	`monix.eval.Task`	`monix.reactive.Observable[ByteBuffer]`	no
`FetchCatsBackend[F[_]: cats.effect.Concurrent]`	`F`	n/a	no

And a backend for scala-native:

Class	Effect type	Supported stream type	Supports websockets
`CurlBackend`	None (`Identity`)	n/a	no

Finally, there are third-party backends:

- sttp-play-ws for "standard" play-ws (not standalone).
- akkaMonixSttpBackend, an Akka-based backend, but using Monix's `Task` & `Observable`.

4.24 Starting & cleaning up

In case of most backends, you should only instantiate a backend once per application, as a backend typically allocates resources such as thread or connection pools.

When ending the application, make sure to call `backend.close()`, which results in an effect which frees up resources used by the backend (if any). If the effect wrapper for the backend is lazily evaluated, make sure to include it when composing effects!

Note that only resources allocated by the backends are freed. For example, if you use the `AkkaHttpBackend()` the `close()` method will terminate the underlying actor system. However, if you have provided an existing actor system upon backend creation (`AkkaHttpBackend.usingActorSystem`), the `close()` method will be a no-op.

4.25 Synchronous backends

There are several synchronous backend implementations. Sending a request using these backends is a blocking operation, and results in a `sttp.client3.Response[T]`.

4.25.1 Using HttpURLConnection

The default **synchronous** backend, available in the main jar for the JVM.

To use, add an implicit value:

```
import sttp.client3._
val backend = HttpURLConnectionBackend()
```

This backend works with all Scala versions. A Scala 3 build is available as well.

This backend supports host header override, but it has to be enabled by system property:

```
sun.net.http.allowRestrictedHeaders=true
```

4.25.2 Using OkHttp

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "okhttp-backend" % "3.1.3"
```

Create the backend using:

```
import sttp.client3.okhttp.OkHttpSyncBackend

val backend = OkHttpSyncBackend()
```

or, if you'd like to instantiate the OkHttpClient yourself:

```
import sttp.client3.okhttp.OkHttpSyncBackend
import okhttp3._

val okHttpClient: OkHttpClient = ???
val backend = OkHttpSyncBackend.usingClient(okHttpClient)
```

This backend depends on OkHttp and fully supports HTTP/2.

4.25.3 Using HttpClient (Java 11+)

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "httpclient-backend" % "3.1.3"
```

Create the backend using:

```
import sttp.client3.httpclient.HttpClientSyncBackend

val backend = HttpClientSyncBackend()
```

or, if you'd like to instantiate the HttpClient yourself:

```
import sttp.client3.httpclient.HttpClientSyncBackend
import java.net.http.HttpClient
val httpClient: HttpClient = ???
val backend = HttpClientSyncBackend.usingClient(httpClient)
```

This backend is based on the built-in `java.net.http.HttpClient` available from Java 11 onwards, works with all Scala versions. A Scala 3 build is available as well.

Host header override is supported in environments running Java 12 onwards, but it has to be enabled by system property:

```
jdk.httpclient.allowRestrictedHeaders=host
```

4.25.4 Streaming

Synchronous backends don't support non-blocking *streaming*.

4.25.5 Websockets

Only the OkHttp backend supports regular *websockets*.

4.26 Akka backend

This backend is based on akka-http. To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "akka-http-backend" % "3.1.3"
```

A fully **asynchronous** backend. Uses the `Future` effect to return responses. There are also other `Future`-based backends, which don't depend on Akka.

Note that you'll also need an explicit dependency on akka-streams, as akka-http doesn't depend on any specific akka-streams version. So you'll also need to add, for example:

```
"com.typesafe.akka" %% "akka-stream" % "2.6.12"
```

Next you'll need to add create the backend instance:

```
import sttp.client3.akkahttp._
val backend = AkkaHttpBackend()
```

or, if you'd like to use an existing actor system:

```
import sttp.client3.akkahttp._
import akka.actor.ActorSystem

val actorSystem: ActorSystem = ???
val backend = AkkaHttpBackend.usingActorSystem(actorSystem)
```

This backend supports sending and receiving akka-streams streams. The streams capability is represented as `sttp.client3.akkahttp.AkkaStreams`.

To set the request body as a stream:

```
import sttp.capabilities.akka.AkkaStreams
import sttp.client3._

import akka.stream.scaladsl.Source
import akka.util.ByteString

val source: Source[ByteString, Any] = ???

basicRequest
  .streamBody(AkkaStreams)(source)
  .post(uri"...")
```

To receive the response body as a stream:

```
import scala.concurrent.Future
import sttp.capabilities.akka.AkkaStreams
import sttp.client3._
import sttp.client3.akkahttp.AkkaHttpBackend

import akka.stream.scaladsl.Source
import akka.util.ByteString

val backend = AkkaHttpBackend()

val response: Future[Response[Either[String, Source[ByteString, Any]]]] =
  basicRequest
    .post(uri"...")
    .response(asStreamUnsafe(AkkaStreams))
    .send(backend)
```

The akka-http backend support both regular and streaming *websockets*.

4.26.1 Testing

Apart from testing using *the stub*, you can create a backend using any `HttpRequest => Future[HttpResponse]` function, or an akka-http `Route`.

That way, you can "mock" a server that the backend will talk to, without starting any actual server or making any HTTP calls.

If your application provides a client library for its dependants to use, this is a great way to ensure that the client actually matches the routes exposed by your application:

```
import sttp.client3.akkahttp._
import akka.http.scaladsl.server.Route
import akka.actor.ActorSystem

val route: Route = ???
implicit val system: ActorSystem = ???

val backend = AkkaHttpBackend.usingClient(system, http = AkkaHttpClient.
↪stubFromRoute(route))
```

4.26.2 WebSockets

Non-standard behavior:

- akka always automatically responds with a `Pong` to a `Ping` message
- `WebSocketFrame.Ping` and `WebSocketFrame.Pong` frames are ignored; instead, you can configure automatic keep-alive pings

4.26.3 Server-sent events

Received data streams can be parsed to a stream of server-sent events (SSE):

```
import scala.concurrent.Future

import akka.stream.scaladsl.Source

import sttp.capabilities.akka.AkkaStreams
import sttp.client3.akkahttp.AkkaHttpServerSentEvents
import sttp.model.sse.ServerSentEvent
import sttp.client3._

def processEvents(source: Source[ServerSentEvent, Any]): Future[Unit] = ???

basicRequest.response(asStream(AkkaStreams)(stream =>
  processEvents(stream.via(AkkaHttpServerSentEvents.parse))))
```

4.27 Future-based backends

There are several backend implementations which are `scala.concurrent.Future`-based. These backends are **asynchronous**, sending a request is a non-blocking operation and results in a response wrapped in a `Future`.

Apart from the ones described below, also the Akka backend is `Future`-based.

Class	Supported stream type	Websocket support
`AkkaHttpBackend`	`akka.stream.scaladsl.Source[ByteString, Any]`	yes (regular & streaming)
`AsyncHttpClientFutureBackend`	n/a	no
`OkHttpFutureBackend`	n/a	yes (regular)
`HttpClientFutureBackend` (Java11+)	n/a	yes (regular)
`ArmeriaBackend`	n/a	n/a

4.27.1 Using async-http-client

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "async-http-client-backend-future" % "3.1.3"
```

And some imports:

```
import sttp.client3._
import sttp.client3.asynchttpclient.future.AsyncHttpClientFutureBackend
```

This backend depends on async-http-client and uses Netty behind the scenes.

Next you'll need to create the backend instance:

```
val backend = AsyncHttpClientFutureBackend()
```

or, if you'd like to use custom configuration:

```
import org.asynchttpclient.AsyncHttpClientConfig

val config: AsyncHttpClientConfig = ???
val backend = AsyncHttpClientFutureBackend.usingConfig(config)
```

or, if you'd like to use adjust the configuration sttp creates:

```
import org.asynchttpclient.DefaultAsyncHttpClientConfig

val sttpOptions: SttpBackendOptions = SttpBackendOptions.Default
val adjustFunction: DefaultAsyncHttpClientConfig.Builder =>␣
↪DefaultAsyncHttpClientConfig.Builder = ???
val backend = AsyncHttpClientFutureBackend.usingConfigBuilder(adjustFunction,␣
↪sttpOptions)
```

or, if you'd like to instantiate the AsyncHttpClient yourself:

```
import org.asynchttpclient.AsyncHttpClient

val asyncHttpClient: AsyncHttpClient = ???
val backend = AsyncHttpClientFutureBackend.usingClient(asyncHttpClient)
```

4.27.2 Using OkHttp

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "okhttp-backend" % "3.1.3"
```

and some imports:

```
import sttp.client3.okhttp.OkHttpFutureBackend
import scala.concurrent.ExecutionContext.Implicits.global
```

Create the backend using:

```
val backend = OkHttpFutureBackend()
```

or, if you'd like to instantiate the OkHttpClient yourself:

```
import okhttp3.OkHttpClient

val asyncHttpClient: OkHttpClient = ???
val backend = OkHttpFutureBackend.usingClient(asyncHttpClient)
```

This backend depends on OkHttp and fully supports HTTP/2.

4.27.3 Using HttpClient (Java 11+)

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "httpclient-backend" % "3.1.3"
```

and some imports:

```
import sttp.client3.httpclient.HttpClientFutureBackend
import scala.concurrent.ExecutionContext.Implicits.global
```

Create the backend using:

```
val backend = HttpClientFutureBackend()
```

or, if you'd like to instantiate the HttpClient yourself:

```
import java.net.http.HttpClient

val client: HttpClient = ???
val backend = HttpClientFutureBackend.usingClient(client)
```

This backend is based on the built-in `java.net.http.HttpClient` available from Java 11 onwards, works with all Scala versions. A Scala 3 build is available as well.

Host header override is supported in environments running Java 12 onwards, but it has to be enabled by system property:

```
jdk.httpclient.allowRestrictedHeaders=host
```

4.27.4 Using Armeria backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "armeria-backend" % "3.1.3"
```

add imports:

```
import sttp.client3.armeria.ArmeriaBackend
import scala.concurrent.ExecutionContext.Implicits.global
```

create client:

```
val backend = ArmeriaBackend()
```

or, if you'd like to instantiate the WebClient yourself::

```
import com.linecorp.armeria.client.WebClient

val client: WebClient = ???
val backend = ArmeriaBackend.usingClient(client)
```

This backend is build on top of Armeria and doesn't support host header override.

4.27.5 Streaming

The Akka backend supports streaming using akka-streams.

Other backends don't support non-blocking *streaming*.

4.27.6 Websockets

Some of the backends (see above) support regular and streaming *websockets*.

4.28 Monix backends

There are several backend implementations which are `monix.eval.Task`-based. These backends are **asynchronous**. Sending a request is a non-blocking, lazily-evaluated operation and results in a response wrapped in a `Task`.

4.28.1 Using async-http-client

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "async-http-client-backend-monix" % "3.1.3"
```

This backend depends on async-http-client, uses Netty behind the scenes.

Next you'll need to define a backend instance as an implicit value. This can be done in two basic ways:

- by creating a `Task`, which describes how the backend is created, or instantiating the backend directly. In this case, you'll need to close the backend manually
- by creating a `Resource`, which will instantiate the backend and close it after it has been used

A non-comprehensive summary of how the backend can be created is as follows:

```
import sttp.client3.asynchttpclient.monix.AsyncHttpClientMonixBackend
import sttp.client3._

AsyncHttpClientMonixBackend().flatMap { backend => ??? }

// or, if you'd like the backend to be wrapped in cats-effect Resource:
AsyncHttpClientMonixBackend.resource().use { backend => ??? }

// or, if you'd like to use custom configuration:
import org.asynchttpclient.AsyncHttpClientConfig
val config: AsyncHttpClientConfig = ???
AsyncHttpClientMonixBackend.usingConfig(config).flatMap { backend => ??? }

// or, if you'd like to use adjust the configuration sttp creates:
import org.asynchttpclient.DefaultAsyncHttpClientConfig
val sttpOptions: SttpBackendOptions = SttpBackendOptions.Default
val adjustFunction: DefaultAsyncHttpClientConfig.Builder =>␣
↪DefaultAsyncHttpClientConfig.Builder = ???
AsyncHttpClientMonixBackend.usingConfigBuilder(adjustFunction, sttpOptions).flatMap {␣
↪backend => ??? }
```

(continues on next page)

(continued from previous page)

```
// or, if you'd like to instantiate the AsyncHttpClient yourself:
import org.asynchttpclient.AsyncHttpClient
val asyncHttpClient: AsyncHttpClient = ???
val backend = AsyncHttpClientMonixBackend.usingClient(asyncHttpClient)
```

4.28.2 Using OkHttp

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "okhttp-backend-monix" % "3.1.3"
```

Create the backend using:

```
import sttp.client3.okhttp.monix.OkHttpMonixBackend

OkHttpMonixBackend().flatMap { backend => ??? }

// or, if you'd like the backend to be wrapped in cats-effect Resource:
OkHttpMonixBackend.resource().use { backend => ??? }

// or, if you'd like to instantiate the OkHttpClient yourself:
import okhttp3._
val okHttpClient: OkHttpClient = ???
val backend = OkHttpMonixBackend.usingClient(okHttpClient)
```

This backend depends on OkHttp and fully supports HTTP/2.

4.28.3 Using HttpClient (Java 11+)

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "httpclient-backend-monix" % "3.1.3"
```

Create the backend using:

```
import sttp.client3.httpclient.monix.HttpClientMonixBackend

HttpClientMonixBackend().flatMap { backend => ??? }

// or, if you'd like the backend to be wrapped in cats-effect Resource:
HttpClientMonixBackend.resource().use { backend => ??? }

// or, if you'd like to instantiate the HttpClient yourself:
import java.net.http.HttpClient
val httpClient: HttpClient = ???
val backend = HttpClientMonixBackend.usingClient(httpClient)
```

This backend is based on the built-in `java.net.http.HttpClient` available from Java 11 onwards.

Host header override is supported in environments running Java 12 onwards, but it has to be enabled by system property:

```
jdk.httpclient.allowRestrictedHeaders=host
```

4.28.4 Streaming

The Monix backends support streaming. The streams capability is represented as `sttp.client3.impl.monix.MonixStreams`. The type of supported streams in this case is `Observable[ByteBuffer]`. That is, you can set such an observable as a request body (using the async-http-client backend as an example, but any of the above backends can be used):

```
import sttp.capabilities.monix.MonixStreams
import sttp.client3._
import sttp.client3.asynchttpclient.monix._

import monix.reactive.Observable

AsyncHttpClientMonixBackend().flatMap { backend =>
  val obs: Observable[Array[Byte]] =  ???

  basicRequest
    .streamBody(MonixStreams)(obs)
    .post(uri"...")
    .send(backend)
}
```

And receive responses as an observable stream:

```
import sttp.capabilities.monix.MonixStreams
import sttp.client3._
import sttp.client3.asynchttpclient.monix._

import monix.eval.Task
import monix.reactive.Observable
import scala.concurrent.duration.Duration

AsyncHttpClientMonixBackend().flatMap { backend =>
  val response: Task[Response[Either[String, Observable[Array[Byte]]]]] =
    basicRequest
      .post(uri"...")
      .response(asStreamUnsafe(MonixStreams))
      .readTimeout(Duration.Inf)
      .send(backend)
    response
}
```

4.28.5 Websockets

The Monix backend supports both regular and streaming *websockets*.

4.28.6 Server-sent events

Received data streams can be parsed to a stream of server-sent events (SSE):

```
import monix.reactive.Observable
import monix.eval.Task

import sttp.capabilities.monix.MonixStreams
```

(continues on next page)

(continued from previous page)

```
import sttp.client3.impl.monix.MonixServerSentEvents
import sttp.model.sse.ServerSentEvent
import sttp.client3._

def processEvents(source: Observable[ServerSentEvent]): Task[Unit] = ???

basicRequest.response(asStream(MonixStreams)(stream =>
  processEvents(stream.transform(MonixServerSentEvents.parse))))
```

4.29 cats-effect backend

The Cats Effect backend is **asynchronous**. It can be created for any type implementing the `cats.effect.Concurrent` typeclass, such as `cats.effect.IO`. Sending a request is a non-blocking, lazily-evaluated operation and results in a wrapped response. There's a transitive dependency on `cats-effect`.

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "async-http-client-backend-cats" % "3.1.3"
```

You'll need the following imports and implicits to create the backend:

```
import sttp.client3._
import sttp.client3.asynchttpclient.cats.AsyncHttpClientCatsBackend
import cats.effect._

// an implicit `cats.effect.ContextShift` in required to create the backend; here,␣
↪for `cats.effect.IO`:
implicit val cs: ContextShift[IO] = IO.contextShift(scala.concurrent.ExecutionContext.
↪global)
```

This backend depends on async-http-client, uses Netty behind the scenes.

Alternatively, the http4s backend can also be created for a type implementing the cats-effect's `Effect` typeclass, and supports streaming as in fs2.

Next you'll need to define a backend instance. This can be done in two basic ways:

- by creating an effect, which describes how the backend is created, or instantiating the backend directly. In this case, you'll need to close the backend manually
- by creating a `Resource`, which will instantiate the backend and close it after it has been used

A non-comprehensive summary of how the backend can be created is as follows:

```
// the type class instance needs to be provided explicitly (e.g. `cats.effect.IO`).
// the effect type must implement the Concurrent typeclass
AsyncHttpClientCatsBackend[IO]().flatMap { backend => ??? }
```

or, if you'd like to use a custom configuration:

```
import org.asynchttpclient.AsyncHttpClientConfig
val config: AsyncHttpClientConfig = ???
AsyncHttpClientCatsBackend.usingConfig[IO](config).flatMap { backend => ??? }
```

or, if you'd like to use adjust the configuration sttp creates:

```
import org.asynchttpclient.DefaultAsyncHttpClientConfig

val sttpOptions: SttpBackendOptions = SttpBackendOptions.Default
val adjustFunction: DefaultAsyncHttpClientConfig.Builder =>␣
↪DefaultAsyncHttpClientConfig.Builder = ???
AsyncHttpClientCatsBackend.usingConfigBuilder[IO](adjustFunction, sttpOptions).
↪flatMap { backend => ??? }
```

or, if you'd like the backend to be wrapped in cats-effect Resource:

```
AsyncHttpClientCatsBackend.resource[IO]().use { backend => ??? }
```

or, if you'd like to instantiate the AsyncHttpClient yourself:

```
import org.asynchttpclient.AsyncHttpClient

val asyncHttpClient: AsyncHttpClient = ???
val backend = AsyncHttpClientCatsBackend.usingClient[IO](asyncHttpClient)
```

4.29.1 Streaming

This backend doesn't support non-blocking *streaming*.

4.29.2 Websockets

The backend doesn't support *websockets*.

4.30 fs2 backend

The fs2 backend is **asynchronous**. It can be created for any type implementing the `cats.effect.Async` typeclass, such as `cats.effect.IO`. Sending a request is a non-blocking, lazily-evaluated operation and results in a wrapped response. There's a transitive dependency on `cats-effect`.

4.30.1 Using async-http-client

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "async-http-client-backend-fs2" % "3.1.3"
```

And some imports:

```
import sttp.client3.asynchttpclient.fs2.AsyncHttpClientFs2Backend
import cats.effect._
import sttp.client3._

// an implicit `cats.effect.ContextShift` is required to create a concurrent instance␣
↪for `cats.effect.IO`,
// as well as a `cats.effect.Blocker` instance. Note that you'll probably want to use␣
↪a different thread
// pool for blocking.
```

(continues on next page)

(continued from previous page)

```
implicit val cs: ContextShift[IO] = IO.contextShift(scala.concurrent.ExecutionContext.
↪global)
val blocker: Blocker = Blocker.liftExecutionContext(scala.concurrent.ExecutionContext.
↪global)
```

This backend depends on async-http-client and uses Netty behind the scenes.

Next you'll need to define a backend instance as an implicit value. This can be done in two basic ways:

- by creating an effect, which describes how a backend is created, or instantiating the backend directly. In this case, you'll need to close the backend manually
- by creating a `Resource`, which will instantiate the backend and close it after it has been used

A non-comprehensive summary of how the backend can be created is as follows:

```
AsyncHttpClientFs2Backend[IO](blocker).flatMap { backend => ??? }
```

or, if you'd like to use a custom configuration:

```
import org.asynchttpclient.AsyncHttpClientConfig

val config: AsyncHttpClientConfig = ???
AsyncHttpClientFs2Backend.usingConfig[IO](blocker, config).flatMap { backend => ??? }
```

or, if you'd like to use adjust the configuration sttp creates:

```
import org.asynchttpclient.DefaultAsyncHttpClientConfig

val sttpOptions: SttpBackendOptions = SttpBackendOptions.Default
val adjustFunction: DefaultAsyncHttpClientConfig.Builder =>␣
↪DefaultAsyncHttpClientConfig.Builder = ???
AsyncHttpClientFs2Backend.usingConfigBuilder[IO](blocker, adjustFunction,␣
↪sttpOptions).flatMap { backend => ??? }
```

or, if you'd like the backend to be wrapped in cats-effect Resource:

```
AsyncHttpClientFs2Backend.resource[IO](blocker).use { backend => ??? }
```

or, if you'd like to instantiate the AsyncHttpClient yourself:

```
import org.asynchttpclient.AsyncHttpClient

val asyncHttpClient: AsyncHttpClient = ???
val backend = AsyncHttpClientFs2Backend.usingClient[IO](asyncHttpClient, blocker)
```

4.30.2 Using HttpClient (Java 11+)

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "httpclient-backend-fs2" % "3.1.3"
```

And some imports:

```
import sttp.client3.httpclient.fs2.HttpClientFs2Backend
import cats.effect._
import sttp.client3._

// an implicit `cats.effect.ContextShift` is required to create a concurrent instance␣
↪for `cats.effect.IO`,
// as well as a `cats.effect.Blocker` instance. Note that you'll probably want to use␣
↪a different thread
// pool for blocking.
implicit val cs: ContextShift[IO] = IO.contextShift(scala.concurrent.ExecutionContext.
↪global)
val blocker = Blocker.liftExecutionContext(scala.concurrent.ExecutionContext.global)
```

Create the backend using:

```
import sttp.client3.httpclient.fs2.HttpClientFs2Backend
HttpClientFs2Backend[IO](blocker).flatMap { backend => ??? }
```

or, if you'd like the backend to be wrapped in cats-effect Resource:

```
HttpClientFs2Backend.resource[IO](blocker).use { backend => ??? }
```

or, if you'd like to instantiate the HttpClient yourself:

```
import java.net.http.HttpClient
val httpClient: HttpClient = ???
val backend = HttpClientFs2Backend.usingClient[IO](httpClient, blocker)
```

This backend is based on the built-in `java.net.http.HttpClient` available from Java 11 onwards.

Host header override is supported in environments running Java 12 onwards, but it has to be enabled by system property:

```
jdk.httpclient.allowRestrictedHeaders=host
```

4.30.3 Streaming

The fs2 backend supports streaming for any instance of the `cats.effect.Effect` typeclass, such as `cats.effect.IO`. If `IO` is used then the type of supported streams is `fs2.Stream[IO, Byte]`. The streams capability is represented as `sttp.client3.fs2.Fs2Streams`.

Requests can be sent with a streaming body like this:

```
import sttp.capabilities.fs2.Fs2Streams
import sttp.client3._
import sttp.client3.asynchttpclient.fs2.AsyncHttpClientFs2Backend
import fs2.Stream

val effect = AsyncHttpClientFs2Backend[IO](blocker).flatMap { backend =>
  val stream: Stream[IO, Byte] = ???

  basicRequest
    .streamBody(Fs2Streams[IO])(stream)
    .post(uri"...")
    .send(backend)
}
```

Responses can also be streamed:

```
import sttp.capabilities.fs2.Fs2Streams
import sttp.client3.asynchttpclient.fs2.AsyncHttpClientFs2Backend
import fs2.Stream
import scala.concurrent.duration.Duration

val effect = AsyncHttpClientFs2Backend[IO](blocker).flatMap { backend =>
  val response: IO[Response[Either[String, Stream[IO, Byte]]]] =
    basicRequest
      .post(uri"...")
      .response(asStreamUnsafe(Fs2Streams[IO]))
      .readTimeout(Duration.Inf)
      .send(backend)

  response
}
```

4.30.4 Websockets

The fs2 backend supports both regular and streaming *websockets*.

4.30.5 Server-sent events

Received data streams can be parsed to a stream of server-sent events (SSE):

```
import cats.effect._
import fs2.Stream

import sttp.capabilities.fs2.Fs2Streams
import sttp.client3.impl.fs2.Fs2ServerSentEvents
import sttp.model.sse.ServerSentEvent
import sttp.client3._

def processEvents(source: Stream[IO, ServerSentEvent]): IO[Unit] = ???

basicRequest.response(asStream(Fs2Streams[IO])(stream =>
  processEvents(stream.through(Fs2ServerSentEvents.parse))))
```

4.31 Scalaz backend

The Scalaz backend is **asynchronous**. Sending a request is a non-blocking, lazily-evaluated operation and results in a response wrapped in a `scalaz.concurrent.Task`. There's a transitive dependency on `scalaz-concurrent`.

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "async-http-client-backend-scalaz" % "3.1.3"
```

This backend depends on async-http-client and uses Netty behind the scenes.

Next you'll need to add an implicit value:

```
import sttp.client3._
import sttp.client3.asynchttpclient.scalaz.AsyncHttpClientScalazBackend

AsyncHttpClientScalazBackend().flatMap { backend => ??? }

// or, if you'd like to use custom configuration:
import org.asynchttpclient.AsyncHttpClientConfig
val config: AsyncHttpClientConfig = ???

AsyncHttpClientScalazBackend.usingConfig(config).flatMap { backend => ??? }

// or, if you'd like to use adjust the configuration sttp creates:
import org.asynchttpclient.DefaultAsyncHttpClientConfig
val sttpOptions: SttpBackendOptions = SttpBackendOptions.Default
val adjustFunction: DefaultAsyncHttpClientConfig.Builder =>␣
↪DefaultAsyncHttpClientConfig.Builder = ???

AsyncHttpClientScalazBackend.usingConfigBuilder(adjustFunction, sttpOptions).flatMap
↪{ backend => ??? }

// or, if you'd like to instantiate the AsyncHttpClient yourself:
import org.asynchttpclient.AsyncHttpClient
val asyncHttpClient: AsyncHttpClient = ???

val backend = AsyncHttpClientScalazBackend.usingClient(asyncHttpClient)
```

4.31.1 Streaming

This backend doesn't support non-blocking *streaming*.

4.31.2 Websockets

The backend doesn't support *websockets*.

4.32 ZIO backends

The ZIO backends are **asynchronous**. Sending a request is a non-blocking, lazily-evaluated operation and results in a response wrapped in a `zio.Task`. There's a transitive dependency on `zio`.

4.32.1 Using HttpClient (Java 11+)

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "httpclient-backend-zio" % "3.1.3"
```

Create the backend using:

```
import sttp.client3.httpclient.zio.HttpClientZioBackend

HttpClientZioBackend().flatMap { backend => ??? }
```

(continues on next page)

(continued from previous page)

```
// or, if you'd like the backend to be wrapped in a Managed:
HttpClientZioBackend.managed().use { backend => ??? }

// or, if you'd like to instantiate the HttpClient yourself:
import java.net.http.HttpClient
val httpClient: HttpClient = ???
val backend = HttpClientZioBackend.usingClient(httpClient)
```

This backend is based on the built-in `java.net.http.HttpClient` available from Java 11 onwards. The backend is fully non-blocking, with back-pressured websockets.

Host header override is supported in environments running Java 12 onwards, but it has to be enabled by system property:

```
jdk.httpclient.allowRestrictedHeaders=host
```

4.32.2 Using async-http-client

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "async-http-client-backend-zio" % "3.1.3"
```

This backend depends on async-http-client, uses Netty behind the scenes. This backend works with all Scala versions. A Scala 3 build is available as well.

Next you'll need to define a backend instance as an implicit value. This can be done in two basic ways:

- by creating a `Task` which describes how the backend is created, or instantiating the backend directly. In this case, you'll need to close the backend manually
- by creating a `TaskManaged`, which will instantiate the backend and close it after it has been used

A non-comprehensive summary of how the backend can be created is as follows:

```
import sttp.client3._
import sttp.client3.asynchttpclient.zio.AsyncHttpClientZioBackend

AsyncHttpClientZioBackend().flatMap { backend => ??? }

// or, if you'd like the backend to be wrapped in a Managed:
AsyncHttpClientZioBackend.managed().use { backend => ??? }

// or, if you'd like to use custom configuration:
import org.asynchttpclient.AsyncHttpClientConfig
val config: AsyncHttpClientConfig = ???
AsyncHttpClientZioBackend.usingConfig(config).flatMap { backend => ??? }

// or, if you'd like to use adjust the configuration sttp creates:
import org.asynchttpclient.DefaultAsyncHttpClientConfig
val sttpOptions: SttpBackendOptions = SttpBackendOptions.Default
val adjustFunction: DefaultAsyncHttpClientConfig.Builder =>␣
↪DefaultAsyncHttpClientConfig.Builder = ???

AsyncHttpClientZioBackend.usingConfigBuilder(adjustFunction, sttpOptions).flatMap {␣
↪backend => ??? }
```

(continues on next page)

(continued from previous page)

```
// or, if you'd like to instantiate the AsyncHttpClient yourself:
import org.asynchttpclient.AsyncHttpClient
import zio.Runtime
val asyncHttpClient: AsyncHttpClient = ???
val runtime: Runtime[Any] = ???
val backend = AsyncHttpClientZioBackend.usingClient(runtime, asyncHttpClient)
```

4.32.3 ZIO environment

As an alternative to effectfully or resourcefully creating backend instances, ZIO environment can be used. In this case, a type alias is provided for the service definition:

```
package sttp.client3.httpclient.zio
type SttpClient = Has[SttpBackend[Task, ZioStreams with WebSockets]]

// or, when using async-http-client

package sttp.client3.asynchttpclient.zio
type SttpClient = Has[SttpBackend[Task, ZioStreams with WebSockets]]
```

The lifecycle of the `SttpClient` service is described by `ZLayer`s, which can be created using the `.layer`/`.layerUsingConfig`/... methods on `AsyncHttpClientZioBackend` / `HttpClientZioBackend`.

The `SttpClient` companion object contains effect descriptions which use the `SttpClient` service from the environment to send requests or open websockets. This is different from sttp usage with other effect libraries (which use an implicit backend when `.send(backend)` is invoked on the request), but is more in line with how other ZIO services work. For example:

```
import sttp.client3._
import sttp.client3.httpclient.zio._
import zio._
val request = basicRequest.get(uri"https://httpbin.org/get")

val sent: ZIO[SttpClient, Throwable, Response[Either[String, String]]] =
  send(request)
```

4.32.4 Streaming

The ZIO based backends support streaming using zio-streams. The following example is using the `AsyncHttpClientZioBackend` backend, but works similarly with `HttpClientZioBackend`.

The type of supported streams is `Stream[Throwable, Byte]`. The streams capability is represented as `sttp.client3.impl.zio.ZioStreams`. To leverage ZIO environment, use the `SttpClient` object to create request send effects.

Requests can be sent with a streaming body:

```
import sttp.capabilities.zio.ZioStreams
import sttp.client3._
import sttp.client3.httpclient.zio.send
import zio.stream._

val s: Stream[Throwable, Byte] =  ???
```

(continues on next page)

(continued from previous page)

```
val request = basicRequest
  .streamBody(ZioStreams)(s)
  .post(uri"...")

send(request)
```

And receive response bodies as a stream:

```
import sttp.capabilities.zio.ZioStreams
import sttp.client3._
import sttp.client3.httpclient.zio.{SttpClient, send}

import zio._
import zio.stream._

import scala.concurrent.duration.Duration

val request =
  basicRequest
    .post(uri"...")
    .response(asStreamUnsafe(ZioStreams))
    .readTimeout(Duration.Inf)

val response: ZIO[SttpClient, Throwable, Response[Either[String, Stream[Throwable,␣
↪Byte]]]] = send(request)
```

4.32.5 Websockets

The ZIO backend supports both regular and streaming *websockets*.

4.32.6 Testing

The ZIO backends also support a ZIO-familiar way of configuring *stubs* as well. In addition to the usual way of creating a stand-alone stub, you can also define your stubs as effects instead:

```
import sttp.client3._
import sttp.model._
import sttp.client3.httpclient._
import sttp.client3.httpclient.zio._
import sttp.client3.httpclient.zio.stubbing._

val stubEffect = for {
  _ <- whenRequestMatches(_.uri.toString.endsWith("c")).thenRespond("c")
  _ <- whenRequestMatchesPartial { case r if r.method == Method.POST => Response.ok("b
↪") }
  _ <- whenAnyRequest.thenRespond("a")
} yield ()

val responseEffect = stubEffect *> send(basicRequest.get(uri"http://example.org/a")).
↪map(_.body)

responseEffect.provideLayer(HttpClientZioBackend.stubLayer) // Task[Either[String,␣
↪String]]
```

The `whenRequestMatches`, `whenRequestMatchesPartial`, `whenAnyRequest` are effects which require the `SttpClientStubbing` dependency. They enrich the stub with the given behavior.

Then, the `stubLayer` provides both an implementation of the `SttpClientStubbing` dependency, as well as a `SttpClient` which is backed by the stub.

4.32.7 Server-sent events

Received data streams can be parsed to a stream of server-sent events (SSE):

```
import zio._
import zio.stream._

import sttp.capabilities.zio.ZioStreams
import sttp.client3.impl.zio.ZioServerSentEvents
import sttp.model.sse.ServerSentEvent
import sttp.client3._

def processEvents(source: Stream[Throwable, ServerSentEvent]): Task[Unit] = ???

basicRequest.response(asStream(ZioStreams)(stream =>
  processEvents(stream.via(ZioServerSentEvents.parse))))
```

4.33 Http4s backend

This backend is based on http4s (client) and is **asynchronous**. To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "http4s-backend" % "3.1.3"
```

Add some imports as well:

```
import cats.effect._
import sttp.client3.http4s._
import scala.concurrent._

// an implicit `cats.effect.ContextShift` is required to create an instance of `cats.
↪effect.Concurrent`
// for `cats.effect.IO`,  as well as a `cats.effect.Blocker` instance.
// Note that you'll probably want to use a different thread pool for blocking.
implicit val cs: ContextShift[IO] = IO.contextShift(ExecutionContext.global)
val blocker: cats.effect.Blocker = Blocker.liftExecutionContext(ExecutionContext.
↪global)
```

The backend can be created for any type implementing the `cats.effect.ConcurrentEffect` typeclass, such as `cats.effect.IO`. Moreover, an implicit `ContextShift` will have to be in scope as well.

If a blocker instance is not available, a new one can be created, and the resource definition can be chained, e.g. as follows:

```
implicit val cs: ContextShift[IO] = IO.contextShift(scala.concurrent.ExecutionContext.
↪global) // or another instance
Blocker[IO].flatMap(Http4sBackend.usingDefaultBlazeClientBuilder[IO](_)).use {␣
↪implicit backend => ... }
```

Sending a request is a non-blocking, lazily-evaluated operation and results in a wrapped response. There's a transitive dependency on `http4s`.

There are also other cats-effect-based backends, which don't depend on http4s.

Please note that:

- the backend contains an **optional** dependency on `http4s-blaze-client`, to provide the `Http4sBackend.usingBlazeClientBuilder` and `Http4sBackend.usingDefaultBlazeClientBuilder` methods. This makes the client usable with other http4s client implementations, without the need to depend on blaze.
- the backend does not support `SttpBackendOptions`,that is specifying proxy settings (proxies are not implemented in http4s, see this issue), as well as configuring the connect timeout
- the backend does not support the `RequestT.options.readTimeout` option

Instead, all custom timeout configuration should be done by creating a `org.http4s.client.Client[F]`, using e.g. `org.http4s.client.blaze.BlazeClientBuilder[F]` and passing it to the appropriate method of the `Http4sBackend` object.

The backend supports streaming using fs2. For usage details, see the documentation on streaming using fs2.

The backend doesn't support *websockets*.

4.34 Twitter future (Finagle) backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "finagle-backend" % "3.1.3"
```

Next you'll need to add an implicit value:

```
import sttp.client3.finagle.FinagleBackend
val backend = FinagleBackend()
```

This backend depends on finagle, and offers an asynchronous backend, which wraps results in Twitter's `Future`.

Please note that:

- the backend does not support `SttpBackendOptions`, that is specifying proxy settings (proxies are not implemented in http4s, see this issue), as well as configuring the connect timeout
- the backend does not support non-blocking *streaming* or *websockets*.

4.35 JavaScript (Fetch) backend

A JavaScript backend implemented using the Fetch API.

4.35.1 `Future`-based

This is the default backend, available in the main jar for JS. To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %%% "core" % "3.1.3"
```

And create the backend instance:

```
val backend = FetchBackend()
```

Timeouts are handled via the new AbortController class. As this class only recently appeared in browsers you may need to add a polyfill.

As browsers do not allow access to redirect responses, if a request sets `followRedirects` to false then a redirect will cause the response to return an error.

Note that `Fetch` does not pass cookies by default. If your request needs cookies then you will need to pass a `FetchOptions` instance with `credentials` set to either `RequestCredentials.same-origin` or `RequestCredentials.include` depending on your requirements.

4.35.2 Monix-based

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %%% "monix" % "3.1.3"
```

And create the backend instance:

```
val backend = FetchMonixBackend()
```

4.35.3 cats-effect-based

Any effect implementing the cats-effect `Concurrent` typeclass can be used. To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %%% "cats" % "3.1.3"
```

And create the backend instance:

```
val backend = FetchCatsBackend[IO]()
```

4.35.4 Node.js

Using `FetchBackend` is possible with node-fetch module.

```
npm install --save node-fetch
```

It has to be loaded into your runtime. This can be done in your main method as such:

```
val g = scalajs.js.Dynamic.global.globalThis

val nodeFetch = g.require("node-fetch")

g.fetch = nodeFetch
g.Headers = nodeFetch.Headers
g.Request = nodeFetch.Request
```

4.35.5 Streaming

Streaming support is provided via `FetchMonixBackend`. Note that streaming support on Firefox is hidden behind a flag, see ReadableStream for more information.

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %%% "monix" % "3.1.3"
```

An example of streaming a response:

```
import sttp.client3._
import sttp.client3.impl.monix._

import java.nio.ByteBuffer
import monix.eval.Task
import monix.reactive.Observable

val backend = FetchMonixBackend()

val response: Task[Response[Observable[ByteBuffer]]] =
  sttp
    .post(uri"...")
    .response(asStreamUnsafe(MonixStreams))
    .send(backend)
```

Note: Currently no browsers support passing a stream as the request body. As such, using the `Fetch` backend with a streaming request will result in it being converted into an in-memory array before being sent. Response bodies are returned as a "proper" stream.

4.36 Curl backend

A Scala Native backend implemented using Curl.

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %%% "core" % "3.1.3"
```

and initialize one of the backends:

```
val backend = CurlBackend()
val tryBackend = CurlTryBackend()
```

You need to have an environment with Scala Native setup with additionally installed `libcrypto` (included in OpenSSL) and `curl` in version `7.56.0` or newer.

4.37 Opentracing backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "opentracing-backend" % "3.1.3"
```

This backend depends on opentracing, a standardized set of api for distributed tracing.

The opentracing backend wraps any other backend, but it's useless without a concrete distributed tracing implementation. To obtain instance of opentracing backend:

```
OpenTracingBackend(wrappedBackend, tracer)
```

Where tracer is an interface which can be implemented by any compatible library. See examples below.

The backend obtains the current trace context using default spans's propagation mechanisms.

There is an additional method exposed to override default operation id:

```
import sttp.client3._
import sttp.client3.opentracing.OpenTracingBackend._

basicRequest
  .get(???)
  .tagWithOperationId("register-user")
```

There is an additional method exposed to customize generated span:

```
import sttp.client3._
import sttp.client3.opentracing.OpenTracingBackend._

basicRequest
  .get(???)
  .tagWithTransformSpan(_.setTag("custom-tag", "custom-value").setOperationName("new-
↪name").log("my-event"))
```

4.37.1 Integration with jaeger

Using with jaeger tracing

Add following dependency:

```
libraryDependencies += "io.jaegertracing" % "jaeger-client" % "1.5.0"
```

Create an instance of tracer:

```
import io.opentracing.Tracer
import io.jaegertracing.Configuration
import io.jaegertracing.Configuration.ReporterConfiguration
import io.jaegertracing.Configuration.SamplerConfiguration

def initTracer(serviceName: String ): Tracer = {
  val samplerConfig = SamplerConfiguration.fromEnv().withType("const").withParam(1)
  val reporterConfig = ReporterConfiguration.fromEnv().withLogSpans(true)
  val config = new Configuration(serviceName).withSampler(samplerConfig)
                   .withReporter(reporterConfig)
  config.getTracer()
}
```

For more details about integration with jaeger click here

4.37.2 Integration with brave

Using with brave tracing

Add following dependency:

```
libraryDependencies += "io.opentracing.brave" % "brave-opentracing" % "1.0.0"
// and for integrationw with okHttp:
libraryDependencies += "io.zipkin.reporter2" % "zipkin-sender-okhttp3" % "2.16.3"
```

Create an instance of tracer:

```
import io.opentracing.Tracer
import zipkin2.reporter.AsyncReporter
import zipkin2.reporter.okhttp3.OkHttpSender
import brave.propagation.{ExtraFieldPropagation, B3Propagation}
import brave.Tracing
import brave.opentracing.BraveTracer
import java.util.Arrays

def initTracer(zipkinUrl: String, serviceName: String): Tracer = {
  // Configure a reporter, which controls how often spans are sent
  val sender = OkHttpSender.create(zipkinUrl)
  val spanReporter = AsyncReporter.create(sender)

  // If you want to support baggage, indicate the fields you'd like to
  // whitelist, in this case "country-code" and "user-id". On the wire,
  // they will be prefixed like "baggage-country-code"
  val propagationFactory = ExtraFieldPropagation.newFactoryBuilder(B3Propagation.
↪FACTORY)
                                  .addPrefixedFields("baggage-",
                                          Arrays.asList("country-code", "user-id"))
                                  .build()

  // Now, create a Brave tracing component with the service name you want to see in
  // Zipkin (the dependency is io.zipkin.brave:brave).
  val braveTracing = Tracing.newBuilder()
                        .localServiceName(serviceName)
                        .propagationFactory(propagationFactory)
                        .spanReporter(spanReporter)
                        .build()

  // use this to create an OpenTracing Tracer
  BraveTracer.create(braveTracing)
}
```

For more details about integration with brave click here

4.38 zio-telemetry opentracing backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "zio-telemetry-opentracing-backend" % "3.1.3"
```

This backend depends on zio-opentracing.

The opentracing backend wraps a `Task` based ZIO backend and yields a backend of type `SttpBackend[RIO[OpenTracing, *], Nothing, WS_HANDLER]`. The yielded effects are of type `RIO[OpenTracing, *]` which mean they can be a child of a other span created in your ZIO program.

Here's how you construct `ZioTelemetryOpenTracingBackend`. I would recommend wrapping this is in `ZLayer`

```
new ZioTelemetryOpenTracingBackend(zioBackend)
```

Additionally you can add tags per request by supplying a `ZioTelemetryOpenTracingTracer`

```
import sttp.client3._
import zio._
import zio.telemetry.opentracing._
import sttp.client3.ziotelemetry.opentracing._

implicit val zioBackend: SttpBackend[Task, Any] = ???

def sttpTracer: ZioTelemetryOpenTracingTracer = new ZioTelemetryOpenTracingTracer {
    def before[T](request: Request[T, Nothing]): RIO[OpenTracing, Unit] =
      OpenTracing.tag("span.kind", "client") *>
      OpenTracing.tag("http.method", request.method.method) *>
      OpenTracing.tag("http.url", request.uri.toString()) *>
      OpenTracing.tag("type", "ext") *>
      OpenTracing.tag("subtype", "http")

    def after[T](response: Response[T]): RIO[OpenTracing, Unit] =
      OpenTracing.tag("http.status_code", response.code.code)
}

ZioTelemetryOpenTracingBackend(zioBackend, sttpTracer)
```

4.39 Prometheus backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "prometheus-backend" % "3.1.3"
```

and some imports:

```
import sttp.client3.prometheus._
```

This backend depends on Prometheus JVM Client. Keep in mind this backend registers histograms and gathers request times, but you have to expose those metrics to Prometheus e.g. using prometheus-akka-http.

The Prometheus backend wraps any other backend, for example:

```
import sttp.client3.akkahttp._
val backend = PrometheusBackend(AkkaHttpBackend())
```

It gathers request execution times in `Histogram`. It uses by default `sttp_request_latency` name, defined in `PrometheusBackend.DefaultHistogramName`. It is possible to define custom histograms name by passing function mapping request to histogram name:

```
import sttp.client3.akkahttp._
val backend = PrometheusBackend(
  AkkaHttpBackend(),
  requestToHistogramNameMapper = request => Some(HistogramCollectorConfig(request.uri.
↪host.getOrElse("example.com")))
)
```

You can disable request histograms by passing `None` returning function:

```
import sttp.client3.akkahttp._
val backend = PrometheusBackend(AkkaHttpBackend(), requestToHistogramNameMapper = _ =>
↪ None)
```

This backend also offers `Gauge` with currently in-progress requests number. It uses by default `sttp_requests_in_progress` name, defined in `PrometheusBackend.DefaultRequestsInProgressGaugeName`. It is possible to define custom gauge name by passing function mapping request to gauge name:

```
import sttp.client3.akkahttp._
val backend = PrometheusBackend(
  AkkaHttpBackend(),
  requestToInProgressGaugeNameMapper = request => Some(CollectorConfig(request.uri.
↪host.getOrElse("example.com")))
)
```

You can disable request in-progress gauges by passing `None` returning function:

```
import sttp.client3.akkahttp._
val backend = PrometheusBackend(AkkaHttpBackend(), requestToInProgressGaugeNameMapper␣
↪= _ => None)
```

4.40 Logging

The `sttp.client3.logging.LoggingBackend` can log requests and responses which end successfully or with an exception. It can be created given:

- a `sttp.client3.logging.Logger`, which is an integration point with logging libraries. Two such integration that are available with sttp-client is slf4j and scribe (see below), but custom ones can be easily added.
- a `sttp.client3.logging.Log`, which constructs messages and performs logging actions. A custom implementation can be provided to change the message content or use dynamic log levels.

By default, the following options are exposed:

- `includeTiming` - should the duration of the request be included in the log message
- `beforeCurlInsteadOfShow` - before sending a request, instead of a summary of the request to be sent, log the curl command which corresponds to the request
- `logRequestBody` - should the request body be logged before sending the request (if the request body can be logged)
- `logRequestHeaders` - should the non-sensitive request headers be logged before sending the request
- `logResponseBody` - should the response body be logged after receiving a response to the request (if the response body can be replayed)
- `logResponseHeaders` - should the non-sensitive response headers be logged

The messages are by default logged on these levels:

- `DEBUG` before the request is sent
- `DEBUG` when a request completes successfully (with a 1xx/2xx status code)
- `WARN` when a request completes successfully (with a 4xx/5xx status code)
- `ERROR` when there's an exception when sending a request

Log levels can be configured when creating the `LoggingBackend`, or specified independently in a custom implementation of `Log`.

4.40.1 Using slf4j

To use the slf4j logging backend wrapper, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "slf4j-backend" % "3.1.3"
```

There are three backend wrappers available, which log request & response information using a slf4j `Logger`. To see the logs, you'll need to use an slf4j-compatible logger implementation, e.g. logback, or use a binding, e.g. log4j-slf4j.

Example usage:

```
import sttp.client3._
import sttp.client3.logging.slf4j.Slf4jLoggingBackend

val backend = Slf4jLoggingBackend(HttpURLConnectionBackend())
basicRequest.get(uri"https://httpbin.org/get").send(backend)
```

To create a customised logging backend, see the section on custom backends.

4.40.2 Using scribe

To use the scribe logging backend wrapper, add the following dependency to your project:

```
"com.softwaremill.sttp.client3" %% "scribe-backend" % "3.1.3"
```

4.41 Custom backends

It is also entirely possible to write custom backends (if doing so, please consider contributing!) or wrap an existing one. One can even write completely generic wrappers for any delegate backend, as each backend comes equipped with a monad for the used effect type. This brings the possibility to `map` and `flatMap` over responses.

Possible use-cases for wrapper-backend include:

- logging
- capturing metrics
- request signing (transforming the request before sending it to the delegate)

See also the section on *resilience* which covers topics such as retries, circuit breaking and rate limiting.

4.41.1 Request tagging

Each request contains a `tags: Map[String, Any]` map. This map can be used to tag the request with any backend-specific information, and isn't used in any way by sttp itself.

Tags can be added to a request using the `def tag(k: String, v: Any)` method, and read using the `def tag(k: String): Option[Any]` method.

Backends, or backend wrappers can use tags e.g. for logging, passing a metric name, using different connection pools, or even different delegate backends.

4.41.2 Listener backend

The `sttp.client3.listener.ListenerBackend` can make it easier to create backend wrappers which need to be notified about request lifecycle events: when a request is started, and when it completes either successfully or with an exception. This is possible by implementing a `sttp.client3.listener.RequestListener`. This is how e.g. the slf4j backend is implemented.

A request listener can associate a value with a request, which will then be passed to the request completion notification methods.

A side-effecting request listener, of type `RequestListener[Identity, L]`, can be lifted to a request listener `RequestListener[F, L]` given a `MonadError[F]`, using the `RequestListener.lift` method.

4.41.3 Backend wrappers and redirects

By default redirects are handled at a low level, using a wrapper around the main, concrete backend: each of the backend factory methods, e.g. `HttpURLConnectionBackend()` returns a backend wrapped in `FollowRedirectsBackend`.

This causes any further backend wrappers to handle a request which involves redirects as one whole, without the intermediate requests. However, wrappers which collects metrics, implements tracing or handles request retries might want to handle every request in the redirect chain. This can be achieved by layering another `FollowRedirectsBackend` on top of the wrapper. Only the top-level follow redirects backend will handle redirects, other follow redirect wrappers (at lower levels) will be disabled.

For example:

```
import sttp.capabilities.Effect
import sttp.client3._
import sttp.monad.MonadError

class MyWrapper[F[_], P] private (delegate: SttpBackend[F, P])
  extends SttpBackend[F, P] {

  def send[T, R >: P with Effect[F]](request: Request[T, R]): F[Response[T]] = ???

  def close(): F[Unit] = ???

  def responseMonad: MonadError[F] = ???
}

object MyWrapper {
  def apply[F[_], P](
    delegate: SttpBackend[F, P]): SttpBackend[F, P] = {
    // disables any other FollowRedirectsBackend-s further down the delegate chain
```

(continues on next page)

(continued from previous page)

```
    new FollowRedirectsBackend(new MyWrapper(delegate))
  }
}
```

4.41.4 Logging backend wrapper

A good example on how to implement a logging backend wrapper is the logging backend wrapper implementation. It uses the `ListenerBackend` to get notified about request lifecycle events.

To adjust the logs to your needs, or to integrate with your logging framework, simply copy the code and modify as needed.

4.41.5 Example metrics backend wrapper

Below is an example on how to implement a backend wrapper, which sends metrics for completed requests and wraps any `Future`-based backend:

```
import sttp.capabilities.Effect
import sttp.client3._
import sttp.client3.akkahttp._
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util._
// the metrics infrastructure
trait MetricsServer {
  def reportDuration(name: String, duration: Long): Unit
}

class CloudMetricsServer extends MetricsServer {
  override def reportDuration(name: String, duration: Long): Unit = ???
}

// the backend wrapper
class MetricWrapper[P](delegate: SttpBackend[Future, P],
                       metrics: MetricsServer)
    extends DelegateSttpBackend[Future, P](delegate) {

  override def send[T, R >: P with Effect[Future]](request: Request[T, R]):␣
↪Future[Response[T]] = {
    val start = System.currentTimeMillis()

    def report(metricSuffix: String): Unit = {
      val metricPrefix = request.tag("metric").getOrElse("?")
      val end = System.currentTimeMillis()
      metrics.reportDuration(metricPrefix + "-" + metricSuffix, end - start)
    }

    delegate.send(request).andThen {
      case Success(response) if response.is200 => report("ok")
      case Success(response)                   => report("notok")
      case Failure(t)                          => report("exception")
    }
  }
}
```

(continues on next page)

(continued from previous page)

```
// example usage
val backend = new MetricWrapper(
  AkkaHttpBackend(),
  new CloudMetricsServer()
)

basicRequest
  .get(uri"http://company.com/api/service1")
  .tag("metric", "service1")
  .send(backend)
```

See also the Prometheus backend for an example implementation.

4.41.6 Example retrying backend wrapper

Handling retries is a complex problem when it comes to HTTP requests. When is a request retryable? There are a couple of things to take into account:

- connection exceptions are generally good candidates for retries
- only idempotent HTTP methods (such as `GET`) could potentially be retried
- some HTTP status codes might also be retryable (e.g. `500 Internal Server Error` or `503 Service Unavailable`)

In some cases it's possible to implement a generic retry mechanism; such a mechanism should take into account logging, metrics, limiting the number of retries and a backoff mechanism. These mechanisms could be quite simple, or involve e.g. retry budgets (see Finagle's documentation on retries). In sttp, it's possible to recover from errors using the `responseMonad`. A starting point for a retrying backend could be:

```
import sttp.capabilities.Effect
import sttp.client3._

class RetryingBackend[F[_], P](
    delegate: SttpBackend[F, P],
    shouldRetry: RetryWhen,
    maxRetries: Int)
    extends DelegateSttpBackend[F, P](delegate) {

  override def send[T, R >: P with Effect[F]](request: Request[T, R]): F[Response[T]]␣
↪= {
    sendWithRetryCounter(request, 0)
  }

  private def sendWithRetryCounter[T, R >: P with Effect[F]](
    request: Request[T, R], retries: Int): F[Response[T]] = {

    val r = responseMonad.handleError(delegate.send(request)) {
      case t if shouldRetry(request, Left(t)) && retries < maxRetries =>
        sendWithRetryCounter(request, retries + 1)
    }

    responseMonad.flatMap(r) { resp =>
      if (shouldRetry(request, Right(resp)) && retries < maxRetries) {
        sendWithRetryCounter(request, retries + 1)
```

(continues on next page)

(continued from previous page)

```
      } else {
        responseMonad.unit(resp)
      }
    }
  }
}
```

4.41.7 Example backend with circuit breaker

> "When a system is seriously struggling, failing fast is better than making clients wait."

There are many libraries that can help you achieve such a behavior: hystrix, resilience4j, akka's circuit breaker or monix catnap to name a few. Despite some small differences, both their apis and functionality are very similar, that's why we didn't want to support each of them explicitly.

Below is an example on how to implement a backend wrapper, which integrates with circuit-breaker module from resilience4j library and wraps any backend:

```
import io.github.resilience4j.circuitbreaker.{CallNotPermittedException,␣
↪CircuitBreaker}
import sttp.capabilities.Effect
import sttp.client3.{Request, Response, SttpBackend, DelegateSttpBackend}
import sttp.monad.MonadError
import java.util.concurrent.TimeUnit

class CircuitSttpBackend[F[_], P](
    circuitBreaker: CircuitBreaker,
    delegate: SttpBackend[F, P]) extends DelegateSttpBackend[F, P](delegate) {

  override def send[T, R >: P with Effect[F]](request: Request[T, R]): F[Response[T]]␣
↪= {
    CircuitSttpBackend.decorateF(circuitBreaker, delegate.send(request))
  }
}

object CircuitSttpBackend {

  def decorateF[F[_], T](
      circuitBreaker: CircuitBreaker,
      service: => F[T]
  )(implicit monadError: MonadError[F]): F[T] = {
    monadError.suspend {
      if (!circuitBreaker.tryAcquirePermission()) {
        monadError.error(CallNotPermittedException
                           .createCallNotPermittedException(circuitBreaker))
      } else {
        val start = System.nanoTime()
        try {
          monadError.handleError(monadError.map(service) { r =>
            circuitBreaker.onSuccess(System.nanoTime() - start, TimeUnit.NANOSECONDS)
            r
          }) {
            case t =>
              circuitBreaker.onError(System.nanoTime() - start, TimeUnit.NANOSECONDS,␣
↪t)
```

(continues on next page)

(continued from previous page)

```
              monadError.error(t)
          }
        } catch {
          case t: Throwable =>
            circuitBreaker.onError(System.nanoTime() - start, TimeUnit.NANOSECONDS, t)
            monadError.error(t)
        }
      }
    }
  }
}
```

4.41.8 Example backend with rate limiter

> "Prepare for a scale and establish reliability and HA of your service."

Below is an example on how to implement a backend wrapper, which integrates with rate-limiter module from resilience4j library and wraps any backend:

```
import io.github.resilience4j.ratelimiter.RateLimiter
import sttp.capabilities.Effect
import sttp.monad.MonadError
import sttp.client3.{Request, Response, SttpBackend, DelegateSttpBackend}

class RateLimitingSttpBackend[F[_], P](
    rateLimiter: RateLimiter,
    delegate: SttpBackend[F, P]
    )(implicit monadError: MonadError[F]) extends DelegateSttpBackend[F, P](delegate)
↪{

  override def send[T, R >: P with Effect[F]](request: Request[T, R]): F[Response[T]]␣
↪= {
    RateLimitingSttpBackend.decorateF(rateLimiter, delegate.send(request))
  }
}

object RateLimitingSttpBackend {

  def decorateF[F[_], T](
      rateLimiter: RateLimiter,
      service: => F[T]
  )(implicit monadError: MonadError[F]): F[T] = {
    monadError.suspend {
      try {
        RateLimiter.waitForPermission(rateLimiter)
        service
      } catch {
        case t: Throwable =>
          monadError.error(t)
      }
    }
  }
}
```

4.41.9 Example new backend

Implementing a new backend is made easy as the tests are published in the `core` jar file under the `tests` classifier. Simply add the follow dependencies to your `build.sbt`:

```
"com.softwaremill.sttp.client3" %% "core" % "3.1.3" % Test classifier "tests"
```

Implement your backend and extend the `HttpTest` class:

```
import sttp.client3._
import sttp.client3.testing.{ConvertToFuture, HttpTest}
import scala.concurrent.Future

class MyCustomBackendHttpTest extends HttpTest[Future] {
  override implicit val convertToFuture: ConvertToFuture[Future] = ConvertToFuture.
↪future
  override lazy val backend: SttpBackend[Future, Any] = ??? //new MyCustomBackend()
  override def timeoutToNone[T](t: Future[T], timeoutMillis: Int): Future[Option[T]]␣
↪= ???
}
```

You can find a more detailed example in the sttp-vertx repository.

4.41.10 Custom backend wrapper using cats

When implementing a backend wrapper using cats, it might be useful to import:

```
import sttp.client3.impl.cats.implicits._
```

from the cats integration module. The module should be available on the classpath when using the cats async-http-client backend. The object contains implicits to convert a cats `MonadError` into the sttp `MonadError`, as well as a way to map the effects wrapper used with the `.mapK` extension method for the backend.

4.42 Testing

If you need a stub backend for use in tests instead of a "real" backend (you probably don't want to make HTTP calls during unit tests), you can use the `SttpBackendStub` class. It allows specifying how the backend should respond to requests matching given predicates.

You can also create a stub backend using *akka-http routes*.

4.42.1 Creating a stub backend

An empty backend stub can be created using the following ways:

- by calling `.stub` on the "real" base backend's companion object, e.g. `AsyncHttpClientZioBackend.stub` or `HttpClientMonixBackend.stub`
- by using one of the factory methods `SttpBackendStub.synchronous` or `SttpBackendStub.asynchronousFuture`, which return stubs which use the `Identity` or standard Scala's `Future` effects without streaming support
- by explicitly specifying the effect and supported capabilities, e.g. `SttpBackendStub[Task, MonixStreams with WebSockets](TaskMonad)`

- by specifying a fallback/delegate backend, see below

Some code which will be reused among following examples:

```
import sttp.client3._
import sttp.model._
import sttp.client3.testing._
import java.io.File
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

case class User(id: String)
```

4.42.2 Specifying behavior

Behavior of the stub can be specified using a series of invocations of the `whenRequestMatches` and `thenRespond` methods:

```
val testingBackend = SttpBackendStub.synchronous
  .whenRequestMatches(_.uri.path.startsWith(List("a", "b")))
  .thenRespond("Hello there!")
  .whenRequestMatches(_.method == Method.POST)
  .thenRespondServerError()

val response1 = basicRequest.get(uri"http://example.org/a/b/c").send(testingBackend)
// response1.body will be Right("Hello there")

val response2 = basicRequest.post(uri"http://example.org/d/e").send(testingBackend)
```

It is also possible to match requests by partial function, returning a response. E.g.:

```
val testingBackend = SttpBackendStub.synchronous
  .whenRequestMatchesPartial({
    case r if r.uri.path.endsWith(List("partial10")) =>
      Response("Not found", StatusCode.NotFound)

    case r if r.uri.path.endsWith(List("partialAda")) =>
      // additional verification of the request is possible
      assert(r.body == StringBody("z", "utf-8"))
      Response.ok("Ada")
  })

val response1 = basicRequest.get(uri"http://example.org/partial10").
↪send(testingBackend)
// response1.body will be Right(10)

val response2 = basicRequest.post(uri"http://example.org/partialAda").
↪send(testingBackend)
```

This approach to testing has one caveat: the responses are not type-safe. That is, the stub backend cannot match on or verify that the type of the response body matches the response body type requested.

Another way to specify the behaviour is passing response wrapped in the effect to the stub. It is useful if you need to test a scenario with a slow server, when the response should be not returned immediately, but after some time. Example with Futures:

```
val testingBackend = SttpBackendStub.asynchronousFuture
  .whenAnyRequest
  .thenRespondF(Future {
    Thread.sleep(5000)
    Response.ok(Right("OK"))
  })

val responseFuture = basicRequest.get(uri"http://example.org").send(testingBackend)
```

The returned response may also depend on the request:

```
val testingBackend = SttpBackendStub.synchronous
  .whenAnyRequest
  .thenRespondF(req =>
    Response.ok(Right(s"OK, got request sent to ${req.uri.host}"))
  )

val response = basicRequest.get(uri"http://example.org").send(testingBackend)
```

You can define consecutive raw responses that will be served:

```
val testingBackend: SttpBackendStub[Identity, Any] = SttpBackendStub.synchronous
  .whenAnyRequest
  .thenRespondCyclic("first", "second", "third")

basicRequest.get(uri"http://example.org").send(testingBackend)       // Right("OK,␣
↪first")       // Right("OK, first")
basicRequest.get(uri"http://example.org").send(testingBackend)       // Right("OK,␣
↪second")       // Right("OK, second")
basicRequest.get(uri"http://example.org").send(testingBackend)       // Right("OK,␣
↪third")       // Right("OK, third")
basicRequest.get(uri"http://example.org").send(testingBackend)       // Right("OK,␣
↪first")
```

Or multiple `Response` instances:

```
val testingBackend: SttpBackendStub[Identity, Any] = SttpBackendStub.synchronous
  .whenAnyRequest
  .thenRespondCyclicResponses(
    Response.ok[String]("first"),
    Response("error", StatusCode.InternalServerError, "Something went wrong")
  )

basicRequest.get(uri"http://example.org").send(testingBackend)       // code will be␣
↪200       // code will be 200
basicRequest.get(uri"http://example.org").send(testingBackend)       // code will be␣
↪500       // code will be 500
basicRequest.get(uri"http://example.org").send(testingBackend)       // code will be␣
↪200
```

The `sttp.client3.testing` package also contains a utility method to force the body as a string (`forceBodyAsString`) or as a byte array (`forceBodyAsByteArray`), if the body is not a stream or multipart:

```
val testingBackend = SttpBackendStub.synchronous
  .whenRequestMatches(_.forceBodyAsString.contains("Hello, world!"))
  .thenRespond("Hello back!")
```

If the stub is given a request, for which no behavior is stubbed, it will return a failed effect with an `IllegalArgumentException`.

4.42.3 Simulating exceptions

If you want to simulate an exception being thrown by a backend, e.g. a socket timeout exception, you can do so by throwing the appropriate exception instead of the response, e.g.:

```
val testingBackend = SttpBackendStub.synchronous
  .whenRequestMatches(_ => true)
  .thenRespond(throw new SttpClientException.ConnectException(
    basicRequest.get(uri"http://example.com"), new RuntimeException))
```

4.42.4 Adjusting the response body type

If the type of the response body returned by the stub's rules (as specified using the `.whenXxx` methods) doesn't match what was specified in the request, the stub will attempt to convert the body to the desired type. This might be useful when:

- testing code which maps a basic response body to a custom type, e.g. mapping a raw json string using a decoder to a domain type
- reading a classpath resource (which results in an `InputStream`) and requesting a response of e.g. type `String`
- using resource-safe response specifications for streaming and websockets

The following conversions are supported:

- anything to `()` (unit), when the response is ignored
- `InputStream` and `Array[Byte]` to `String`
- `InputStream` and `String` to `Array[Byte]`
- `WebSocketStub` to `WebSocket`
- `WebSocketStub` and `WebSocket` are supplied to the websocket-consuming functions, if the response specification describes such interactions
- `SttpBackendStub.RawStream` is always treated as a raw stream value, and returned when the response should be returned as a stream or consumed using the provided function
- any of the above to custom types through mapped response specifications

4.42.5 Example: returning JSON

For example, if you want to return a JSON response, simply use `.withResponse(String)` as below:

```
val testingBackend = SttpBackendStub.synchronous
  .whenRequestMatches(_ => true)
  .thenRespond(""" {"username": "john", "age": 65 } """)

def parseUserJson(a: Array[Byte]): User = ???

val response = basicRequest.get(uri"http://example.com")
  .response(asByteArrayAlways.map(parseUserJson))
  .send(testingBackend)
```

In the example above, the stub's rules specify that a response with a `String`-body should be returned for any request; the request, on the other hand, specifies that response body should be parsed from a byte array to a custom `User` type. These type don't match, so the `SttpBackendStub` will in this case convert the body to the desired type.

4.42.6 Example: returning a file

If you want to return a file and have a response handler set up like this:

```
val destination = new File("path/to/file.ext")
basicRequest.get(uri"http://example.com").response(asFile(destination))
```

Then set up the stub like this:

```
val fileResponseHandle = new File("path/to/file.ext")
SttpBackendStub.synchronous
  .whenRequestMatches(_ => true)
  .thenRespond(fileResponseHandle)
```

the `File` set up in the stub will be returned as though it was the `File` set up as `destination` in the response handler above. This means that the file from `fileResponseHandle` is not written to `destination`.

If you actually want a file to be written you can set up the stub like this:

```
import org.apache.commons.io.FileUtils
import cats.effect._
import sttp.client3.impl.cats.implicits._
import sttp.monad.MonadAsyncError

implicit val cs: ContextShift[IO] = IO.contextShift(scala.concurrent.ExecutionContext.
↪global)

val sourceFile = new File("path/to/file.ext")
val destinationFile = new File("path/to/file.ext")
SttpBackendStub(implicitly[MonadAsyncError[IO]])
  .whenRequestMatches(_ => true)
  .thenRespondF { _ =>
    FileUtils.copyFile(sourceFile, destinationFile)
    IO(Response(Right(destinationFile), StatusCode.Ok, ""))
  }
```

4.42.7 Delegating to another backend

It is also possible to create a stub backend which delegates calls to another (possibly "real") backend if none of the specified predicates match a request. This can be useful during development, to partially stub a yet incomplete API with which we integrate:

```
val testingBackend =
  SttpBackendStub.withFallback(HttpURLConnectionBackend())
    .whenRequestMatches(_.uri.path.startsWith(List("a")))
    .thenRespond("I'm a STUB!")

val response1 = basicRequest.get(uri"http://api.internal/a").send(testingBackend)
// response1.body will be Right("I'm a STUB")

val response2 = basicRequest.post(uri"http://api.internal/b").send(testingBackend)
```

4.42.8 Testing streams

Streaming responses can be stubbed the same as ordinary values, with one difference. If the stubbed response contains the raw stream, which should be then transformed as described by the response specification, the stub must know that it handles a raw stream. This can be achieved by wrapping the stubbed stream using `SttpBackendStub.RawStream`.

If the response specification is a resource-safe consumer of the stream, the function will only be invoked if the body is a `RawStream` (with the contained value).

Otherwise, the stub can be also configured to return the high-level (already mapped/transformed) response body.

4.42.9 Testing web sockets

Like streams, web sockets can be stubbed as ordinary values, by providing `WebSocket` or `WebSocketStub` instances.

If the response specification is a resource-safe consumer of the web socket, the function will be invoked if the provided stubbed body is a `WebSocket` or `WebSocketStub`.

The stub can be configured to return the high-level (already mapped/transformed) response body.

WebSocketStub

`WebSocketStub` allows easy creation of stub `WebSocket` instances. Such instances wrap a state machine that can be used to simulate simple WebSocket interactions. The user sets initial responses for `receive` calls as well as logic to add further messages in reaction to `send` calls.

For example:

```
import sttp.ws.testing.WebSocketStub
import sttp.ws.WebSocketFrame

val backend = SttpBackendStub.synchronous
val webSocketStub = WebSocketStub
  .initialReceive(
    List(WebSocketFrame.text("Hello from the server!"))
  )
  .thenRespondS(0) {
    case (counter, tf: WebSocketFrame.Text) => (counter + 1, List(WebSocketFrame.
↪text(s"echo: ${tf.payload}")))
    case (counter, _)                       => (counter, List.empty)
  }

backend.whenAnyRequest.thenRespond(webSocketStub)
```

There is a possiblity to add error responses as well. If this is not enough, using a custom implementation of the `WebSocket` trait is recommended.

4.42.10 Verifying, that a request was sent

Using `RecordingSttpBackend` it's possible to capture all interactions in which a backend has been involved.

The recording backend is a *backend wrapper*, and it can wrap any backend, but it's most useful when combine witht the backend stub.

Example usage:

```
import scala.util.Try

val testingBackend = new RecordingSttpBackend(
  SttpBackendStub.synchronous
    .whenRequestMatches(_.uri.path.startsWith(List("a", "b")))
    .thenRespond("Hello there!")
)

val response1 = basicRequest.get(uri"http://example.org/a/b/c").send(testingBackend)
// response1.body will be Right("Hello there")

testingBackend.allInteractions: List[(Request[_, _], Try[Response[_]])]
```

4.43 Timeouts

sttp supports read and connection timeouts:

- Connection timeout - can be set globally (30 seconds by default)
- Read timeout - can be set per request (1 minute by default)

How to use:

```
import sttp.client3._
import scala.concurrent.duration._

// all backends provide a constructor that allows to specify backend options
val backend = HttpURLConnectionBackend(
  options = SttpBackendOptions.connectionTimeout(1.minute))

basicRequest
  .get(uri"...")
  .readTimeout(5.minutes) // or Duration.Inf to turn read timeout off
  .send(backend)
```

4.44 SSL

SSL handling can be customized (or disabled) when creating a backend and is backend-specific.

Depending on the underlying backend's client, you can customize SSL settings as follows:

- `HttpUrlConnectionBackend`: when creating the backend, specify the `customizeConnection: HttpURLConnection => Unit` parameter, and set the hostname verifier & SSL socket factory as required
- akka-http: when creating the backend, specify the `customHttpsContext: Option[HttpsConnectionContext]` parameter. See akka-http docs
- async-http-client: create a custom client and use the `setSSLContext` method
- OkHttp: create a custom client modifying the SSL settings as described on the wiki

4.45 Proxy support

sttp library by default checks for your System proxy properties (docs):

Following settings are checked:

1. `socksProxyHost` and `socksProxyPort` *(default: 1080)*
2. `http.proxyHost` and `http.proxyPort` *(default: 80)*
3. `https.proxyHost` and `https.proxyPort` *(default: 443)*

Settings are loaded **in given order** and the **first existing value** is being used.

Otherwise, proxy values can be specified manually when creating a backend:

```
import sttp.client3._

val backend = HttpURLConnectionBackend(
  options = SttpBackendOptions.httpProxy("some.host", 8080))

basicRequest
  .get(uri"...")
  .send(backend) // uses the proxy
```

Or in case your proxy requires authentication (supported by the JVM backends):

```
import sttp.client3._

SttpBackendOptions.httpProxy("some.host", 8080, "username", "password")
```

4.46 Redirects

By default, sttp follows redirects.

If you'd like to disable following redirects, use the `followRedirects` method:

```
import sttp.client3._

basicRequest.followRedirects(false)
```

If a request has been redirected, the history of all followed redirects is accessible through the `response.history` list. The first response (oldest) comes first. The body of each response will be a `Left(message)` (as the status code is non-2xx), where the message is whatever the server returned as the response body.

4.46.1 Redirecting POST requests

If a `POST` or `PUT` request is redirected, by default it will be sent unchanged to the new address, that is using the original body and method. However, most browsers and some clients issue a `GET` request in such case, without the body.

To enable this behavior, use the `redirectToGet` method:

```
import sttp.client3._

basicRequest.redirectToGet(true)
```

Note that this only affects `301 Moved Permanently` and `302 Found` redirects. `303 See Other` redirects are always converted, while `307 Temporary Redirect` and `308 Permanent Redirect` never.

4.47 Other Scala HTTP clients

- scalaj
- akka-http client
- dispatch
- play ws
- fs2-http
- http4s
- Gigahorse
- RösHTTP
- Requests-Scala

Also, check the comparison by Marco Firrincieli on how to implement a simple request using a number of Scala HTTP libraries.