sttp Documentation

Release 1.0

Adam Warski

Jan 10, 2020

Getting started

Welcome!

sttp client is an open-source library which provides a clean, programmer-friendly API to describe HTTP requests and execute them using one of the wrapped backends, such as akka-http, async-http-client, http4s or OkHttp.

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

```
import sttp.client._

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

// 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")

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

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

// response.unsafeBody: by default read into a String
println(response.unsafeBody)
```

For more examples, see the *usage examples* section. Or explore the features in detail:

CHAPTER 1

Quickstart

The main sttp client API comes in a single jar, with a single transitive dependency on the sttp model. This also includes a default, synchronous backend, which is based on Java's `HttpURLConnection`. For production usages, you'll often want to use an alternate backend (but what's important is that the API remains the same!). See the section on *backends* for additional instructions.

1.1 Using sbt

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

```
"com.softwaremill.sttp.client" %% "core" % "2.0.0-RC6"
```

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

`sttp client` is also available for Scala.js 0.6 and Scala Native. Note that not all modules are compatible and there are no backends that can be used on both.

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.client::core:2.0.0-RC6`
import sttp.client.quick._
quickRequest.get(uri"http://httpbin.org/ip").send()
```

Importing the `quick` object has the same effect as importing `sttp.client._`, plus defining an implicit synchronous backend (`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:

```
import $ivy.`com.softwaremill.sttp.client::okhttp-backend:2.0.0-RC6`
import sttp.client.okhttp.quick._
quickRequest.get(uri"http://httpbin.org/ip").send()
```

1.3 Imports

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

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

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! Or, read on about the *basics of defining requests*.

CHAPTER 2

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 introduction to sttp and sttp streaming & URI interpolators blogs.

2.1 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

2.2 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

CHAPTER 3

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!

CHAPTER 4

Usage examples

4.1 POST a form using the synchronous backend

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client" %% "core" % "2.0.0-RC6")
```

Example code:

```
import sttp.client._

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")

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

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

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

Required dependencies:

```
libraryDependencies ++= List(
  "com.softwaremill.sttp.client" %% "akka-http-backend" % "2.0.0-RC6",
```

(continues on next page)

(continued from previous page)

```
  "com.softwaremill.sttp.client" %% "json4s" % "2.0.0-RC6",
  "org.json4s" %% "json4s-native" % "3.6.0"
)
```

Example code:

```
import sttp.client._
import sttp.client.akkahttp._
import sttp.client.json4s._

import scala.concurrent.ExecutionContext.Implicits.global

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

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

implicit val backend = AkkaHttpBackend()
val response: Future[Response[Either[ResponseError[Exception], HttpBinResponse]]] =
  request.send()

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

4.3 Test an endpoint requiring multiple parameters

Required dependencies:

```
libraryDependencies ++= List("com.softwaremill.sttp.client" %% "core" % "2.0.0-RC6")
```

Example code:

```
import sttp.client._
import sttp.client.testing._

implicit 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()
    .unsafeBody)
```

(continues on next page)

(continued from previous page)

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

CHAPTER 5

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` and `Uri`. The `.toString` methods of these classes returns a representation as in a HTTP request/response. See the ScalaDoc for more information.

Constructors of the model classes are mostly private. 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
- `.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 of 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
- `.notValidated(...): 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 `MultiQueryParams`

Example with objects:

```
import sttp.client._

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

(continues on next page)

(continued from previous page)

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

Example with traits:

```
import sttp.client._

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

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

For more information see

- https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
- https://en.wikipedia.org/wiki/Media_type
- https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

CHAPTER 6

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 of 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.

6.1 URI interpolator

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

```
import sttp.client._

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, with the exception of the `:` in the host part, which is commonly used to pass in both the host and port:

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

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

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

6.2 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")

val u1 = uri"http://example.com?p1=$v1&p2=v2"
assert(u1.toString == "http://example.com?p2=v2")

val u2 = uri"http://$v1.$v2.example.com"
assert(u2.toString == "http://v2.example.com")

val u3 = uri"http://example.com#$v1"
assert(u3.toString == "http://example.com")
```

6.3 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")
val u4 = uri"http://example.com?$ps&p3=p4"
assert(u4.toString == "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 ps = List("a", "b", "c")
val u5 = uri"http://example.com/$ps"
assert(u5.toString == "http://example.com/a/b/c")
```

6.4 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.

6.5 All features combined

A fully-featured example:

```
import sttp.client._
val secure = true
val scheme = if (secure) "https" else "http"
val subdomains = List("sub1", "sub2")
val vx = Some("y z")
val params = Map("a" -> 1, "b" -> 2)
val jumpTo = Some("section2")
uri"$scheme://$subdomains.example.com?x=$vx&$params#$jumpTo"

// generates:
// https://sub1.sub2.example.com?x=y+z&a=1&b=2#section2
```

CHAPTER 7

Request definition basics

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

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

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()` method is invoked.

7.1 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 out-of the box.

To invoke the `send()` method on a request description, an implicit value of type `SttpBackend` needs to be in scope:

```
implicit val backend = HttpURLConnectionBackend()

val response: Response[String] = request.send()
```

The default backend doesn't wrap the response into any container, but other asynchronous backends might do so. 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*.

7.2 Starting requests

sttp provides two starting 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.

7.3 Debugging requests

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

For example converting given request:

```
basicRequest.get(uri"http://httpbin.org/ip").toCurl
```

will result in following curl command:

```
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.

CHAPTER 8

Headers

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

```
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:

```
def header(h: Header, replaceExisting: Boolean = false)
def header(k: String, v: String)
def header(k: String, v: String, replaceExisting: Boolean)
def headers(hs: Map[String, String])
def headers(hs: (String, String)*)
def headers(hs: Header*)
```

8.1 Common headers

For some common headers, dedicated methods are provided:

```
def contentType(ct: String)
def contentType(ct: String, encoding: String)
def contentLength(l: Long)
def acceptEncoding(encoding: String)
```

See also documentation on setting *cookies* and *authentication*.

CHAPTER 9

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:

```
def cookie(nv: (String, String))
def cookie(n: String, v: String)
def cookies(nvs: (String, String)*)
def cookies(cs: Iterable[Cookie])
```

9.1 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:

```
// Method signature
def cookies(r: Response[_])

// Usage
basicRequest.cookies(response)
```

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

```
def cookies(cs: Seq[Cookie])
```

CHAPTER 10

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:

```
basicRequest.auth.basic(username, password)
```

A bearer token can be added using:

```
basicRequest.auth.bearer(token)
```

10.1 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[R, S, WS_HANDLER] = ???
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
- it doesn't work in scala-native

CHAPTER 11

Setting the request body

11.1 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:

```
basicRequest.body("Hello, world!")
```

It is also possible to use a different character encoding:

```
def body(b: String)
def body(b: String, encoding: String)
```

11.2 Binary data

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

```
def body(b: Array[Byte])
def body(b: ByteBuffer)
def body(b: 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.

11.3 Uploading files

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

```
def body(f: File)
def body(b: Path)
```

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.

11.4 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:

```
def body(fs: Map[String, String])
def body(fs: Map[String, String], encoding: String)
def body(fs: (String, String)*)
def body(fs: Seq[(String, String)], encoding: String)
```

11.5 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:

```
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", Some("application/csv"))
}

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

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

CHAPTER 12

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.client` 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:

```
def multipartBody(ps: Seq[Multipart])
def multipartBody(p1: Multipart, ps: Multipart*)
```

For example:

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

12.1 Customising part meta-data

For each part, an optional filename can be specified, as well as a custom content type and additional headers. The following methods are available on `Multipart` instances:

```
case class Multipart {
  def fileName(v: String): Multipart
  def contentType(v: String): Multipart
  def header(k: String, v: String): Multipart
}
```

For example:

```
basicRequest.multipartBody(
  multipartFile("logo", logoFile).fileName("logo.jpg").contentType("image/jpg"),
  multipartFile("text", docFile).fileName("text.doc")
)
```

CHAPTER 13

Streaming

Some backends (see *backends summary*) support streaming bodies. 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*.

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

```
import sttp.client._
import sttp.client.akkahttp._

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

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

basicRequest
  .streamBody(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.

CHAPTER 14

Websockets

Apart from streaming, backends (see *backends summary*) can also optionally support websockets. Websocket requests are described exactly the same as regular requests, starting with `basicRequest`, adding headers, specifying the request method and uri.

The difference is that `openWebsocket(handler)` should be called instead of `send()`, given an instance of a backend-specific websocket handler. Refer to documentation of individual backends for details on how to instantiate the handler.

If creating the websocket handler is a side-effecting operation (and the handler is wrapped with an effects wrapper), the `openWebsocketF(handler)` can be used.

After opening a websocket, a `WebSocketResponse` instance is returned, wrapped in a backend-specific effects wrapper, such as `Future`, `IO`, `Task` or no wrapper for synchronous backends. If the protocol upgrade hasn't been successful, the request will fail with an error (represented as an exception or a failed effects wrapper).

In case of success, `WebSocketResponse` contains:

- the headers returned when opening the websocket
- a handler-specific and backend-specific value, which can be used to interact with the websocket, or somehow representing the result of the connection

14.1 Low-level and high-level websocket handlers

Each backend which supports websockets, does so through a backend-specific websocket handler. Depending on the backend, this can be an implementation of a "low-level" Java listener interface (as in *async-http-client*, *OkHttp* and *HttpClient*), a Scala stream (as in *akka-http* and *fs2*), or some other other approach.

Additionally, some backends, on top of the "low-level" Java listeners also offer a higher-level, more "functional" approach to websockets. This is done by passing a specific handler instance when opening the websocket; refer to the documentation of individual backends for details.

Note: The listeners created by the high-level handlers internally buffer incoming websocket events. In some implementations, when creating the handler, a bound can be specified for the size of the buffer. If the bound is specified and the buffer fills up (as can happen if the messages are not received, or processed slowly), the websocket will error and close. Otherwise, the buffer will potentially take up all available memory.

When the websocket is open, the response will contain an instance of `sttp.client.ws.WebSocket[F]`, where `F` is the backend-specific effects wrapper, such as `IO` or `Task`. This interface contains two methods, both of which return computations wrapped in the effects wrapper `F` (which typically is lazily-evaluated description of a side-effecting, asynchronous process):

- `def receive: F[Either[WebSocketEvent.Close, WebSocketFrame.Incoming]]` which will complete once a message is available, and return either information that the websocket has been closed, or the incoming message
- `def send(f: WebSocketFrame, isContinuation: Boolean = false): F[Unit]`, which should be used to send 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`.

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

If there's an error, a failed effects wrapper will be returned, containing one of the `sttp.client.ws.WebSocketError` exceptions, or a backend-specific exception.

Example usage with the Monix variant of the async-http-client backend:

```
import monix.eval.Task
import sttp.client._
import sttp.client.ws.{WebSocket, WebSocketResponse}
import sttp.model.ws.WebSocketFrame

val response: Task[WebSocketResponse[WebSocket[Task]]] = basicRequest
  .get(uri"wss://echo.websocket.org")
  .openWebsocketF(MonixWebSocketHandler())

response.flatMap { r =>
  val ws: WebSocket[Task] = r.result
  val send = ws.send(WebSocketFrame.text("Hello!"))
  val receive = ws.receiveText().flatMap(t => Task(println(s"RECEIVED: $t")))
  val close = ws.close()
  send.flatMap(_ => receive).flatMap(_ => close)
}
```

The high-level handler can be further wrapped to obtain a stream, see the section on fs2 websockets in *async-http-client*.

CHAPTER 15

The type of request definitions

All request definitions have type `RequestT[U, T, S]` (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.
- `S` specifies the stream type that this request uses. Most of the time this will be `Nothing`, meaning that this request does not send a streaming body or receive a streaming response. So most of the time you can just ignore that parameter. But, if you are using a streaming backend and want to send/receive a stream, the `.streamBody` or `response(asStream[S])` will change the type parameter.

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

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

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

```
basicRequest: PartialRequest[Either[String, String], Nothing]
```

CHAPTER 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, the response will be returned in a wrapper. 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, `Id`, 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 an error will be represented in the wrapper (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* and/or handling non 2xx responses for details on how such cases are handled.

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.

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:

```
def header(h: String): Option[String]
def headers(h: String): Seq[String]
```

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

```
def contentType: Option[String]
def contentLength: Option[Long]
```

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

```
def cookies: Seq[Cookie]
```

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.

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*.

CHAPTER 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 expected (non 2xx).
- `Right(body)` if the request is successful and the response code is expected (2xx).

How the response body will be read is part of the request definition, 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, S]` property.

17.1 Basic response specifications

To conveniently specify how to deserialize the response body, a number of `asXxx` methods are available. They can be used to provide a value for the request definition's `response` modifier:

```
sttp.response(asByteArray)
```

When the above request is completed and sent, it will result in a `Response[Either[String, Array[Byte]]]`. Other possible response specifications are:

```
def ignore: ResponseAs[Unit, Nothing]
def asString: ResponseAs[Either[String, String], Nothing]
def asStringAlways: ResponseAs[String, Nothing]
def asString(encoding: String): ResponseAs[Either[String, String], Nothing]
def asStringAlways(encoding: String): ResponseAs[String, Nothing]
def asByteArray: ResponseAs[Either[String, Array[Byte]], Nothing]
def asByteArrayAlways: ResponseAs[Array[Byte], Nothing]
def asParams: ResponseAs[Either[String, Seq[(String, String)]], Nothing]
```

(continues on next page)

(continued from previous page)

```
def asParams(encoding: String): ResponseAs[Either[String, Seq[(String, String)]],
↪Nothing]
def asFile(file: File): ResponseAs[Either[String, File], Nothing]
def asFileAlways(file: File): ResponseAs[File, Nothing]
def asPath(path: Path): ResponseAs[Either[String, Path], Nothing]
def asPathAlways(path: Path): ResponseAs[Path], Nothing]

def asEither[L, R, S](onError: ResponseAs[L, S], onSuccess: ResponseAs[R, S]):
↪ResponseAs[Either[L, R], S]
def fromMetadata[T, S](f: ResponseMetadata => ResponseAs[T, S]): ResponseAs[T, S]
```

Hence, to discard the response body, simply specify:

```
sttp.response(ignore)
```

And to save the response to a file:

```
sttp.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.

17.2 Custom body deserializers

It's possible to define custom body deserializers by taking any of the built-in response specifications and mapping over them. Each `ResponseAs` instance has `map` and `mapWithMetadata` methods, which can be used to transform it to a specification 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.

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

```
val asInt: ResponseAs[Either[String, Int], Nothing] = asString.map(_.toInt)

sttp
  .response(asInt)
  ...
```

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

```
def parseJson(json: String): Either[JsonError, JsonAST] = ...
val asJson: ResponseAs[Either[JsonError, JsonAST], Nothing] = asStringAlways.
↪map(parseJson)

sttp
  .response(asJson)
  ...
```

For some mapped response specifications available out-of-the-box, see *json support*.

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.

17.3 Streaming

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

```
def asStream[S]: ResponseAs[Either[String, S], S] = ResponseAsStream[S, S]()
def asStreamAlways[S]: ResponseAs[S, S] = ResponseAsStream[S, S]()
```

For example, when using the *akka-http backend*:

```
import sttp.client._
import sttp.client.akkahttp._

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

implicit val sttpBackend = AkkaHttpBackend()

val response: Future[Response[Source[Either[String, ByteString], Any]]] =
  sttp
    .post(uri"...")
    .response(asStream[Source[ByteString, Any]])
    .send()
```

Note: Unlike with non-streaming response handlers, each streaming response should be entirely consumed by client code.

CHAPTER 18

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. It's the backends where all the heavy-lifting is done.

Choosing the right backend depends on a number of factors: if 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`
- if you have Akka in your stack, use `AkkaHttpBackend`
- otherwise, if you are using `Future`, use `AsyncHttpClientFutureBackend`
- finally, if you are using a functional effect wrapper, use one of the "functional" async-http-client backends

Each backend has three type parameters:

- `F[_]`, the effects wrapper for responses. That is, when you invoke `send()` on a request description, do you get a `Response[_]` directly, or is it wrapped in a `Future` or a `Task`?
- `S`, the type of supported streams. If `Nothing`, streaming is not supported. Otherwise, the given type can be used to send request bodies or receive response bodies.
- `WS_HANDLER`, the type of supported websocket handlers. If `NothingT`, websockets are not supported. Otherwise, websocket connections can be opened, given an instance of the handler

Below is a summary of all the backends. See the sections on individual backend implementations for more information.

Class	Response wrapper	Supported stream type	Supported websocket handlers
HttpURLConnectionBackend	None (Id)	n/a	n/a
TryHttpURLConnectionBackend	scala.util.Try	n/a	n/a
AkkaHttpBackend	scala.concurrent.Future	akka.stream.scaladsl.Source[ByteString, Any]	akka.stream.scaladsl.Flow[Message, Message, _]
AsyncHttpClientFutureBackend	scala.concurrent.Future	n/a	sttp.client.asynchttpclient.WebSocketHandler
AsyncHttpClientScalazBackend	scalaz.concurrent.Task	n/a	sttp.client.asynchttpclient.WebSocketHandler
AsyncHttpClientZioBackend	zio.IO	n/a	sttp.client.asynchttpclient.WebSocketHandler
AsyncHttpClientZioStreamsBackend	zio.IO	zio.stream.Stream[Throwable, ByteBuffer]	sttp.client.asynchttpclient.WebSocketHandler
AsyncHttpClientMonixBackend	monix.eval.Task	monix.reactive.Observable[ByteBuffer]	sttp.client.asynchttpclient.WebSocketHandler
AsyncHttpClientCatsBackend	F[_]: cats.effect.Async	n/a	sttp.client.asynchttpclient.WebSocketHandler
AsyncHttpClientFs2Backend	F[_]: cats.effect.Async	fs2.Stream[F, ByteBuffer]	sttp.client.asynchttpclient.WebSocketHandler
OkHttpSyncBackend	None (Id)	n/a	sttp.client.okhttp.WebSocketHandler
OkHttpFutureBackend	scala.concurrent.Future	n/a	sttp.client.okhttp.WebSocketHandler
OkHttpMonixBackend	monix.eval.Task	monix.reactive.Observable[ByteBuffer]	sttp.client.okhttp.WebSocketHandler
Http4sBackend	F[_]: cats.effect.Effect	fs2.Stream[F, Byte]	n/a
HttpClientSyncBackend	None (Id)	n/a	sttp.client.httpclient.WebSocketHandler
HttpClientFutureBackend	scala.concurrent.Future	n/a	sttp.client.httpclient.WebSocketHandler
HttpClientMonixBackend	monix.eval.Task	monix.reactive.Observable[ByteBuffer]	sttp.client.httpclient.WebSocketHandler
FinagleBackend	com.twitter.util.Future	n/a	n/a

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`
- `BraveBackend`, for Zipkin-compatible distributed tracing. See the *dedicated section*.
- `PrometheusBackend`, for gathering Prometheus-format metrics. See the *dedicated section*.

In additional there are also backends for JavaScript:

Class	Response wrapper	Supported stream type	Supported websocket handlers
`FetchBackend`	`scala.concurrent.Future`	n/a	n/a
`FetchMonixBackend`	`monix.eval.Task`	`monix.reactive.Observable[ByteBuffer]`	n/a

and Scala Native:

Class	Response wrapper	Supported stream type	Supported websocket handlers
`CurlBackend`	None (`id`)	n/a	n/a
`CurlTryBackend`	`scala.util.Try`	n/a	n/a

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`.

CHAPTER 19

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.

CHAPTER 20

HttpURLConnection backend

The default **synchronous** backend. Sending a request returns a response wrapped in the identity type constructor, which is equivalent to no wrapper at all.

To use, add an implicit value:

```
implicit val sttpBackend = HttpURLConnectionBackend()
```

CHAPTER 21

akka-http backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "akka-http-backend" % "2.0.0-RC6"
```

This backend depends on akka-http. A fully **asynchronous** backend. Sending a request returns a response wrapped in a `Future`.

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" % "2.5.11"
```

Next you'll need to add an implicit value:

```
implicit val sttpBackend = AkkaHttpBackend()

// or, if you'd like to use an existing actor system:
implicit val sttpBackend = AkkaHttpBackend.usingActorSystem(actorSystem)
```

This backend supports sending and receiving akka-streams streams of type `akka.stream.scaladsl.Source[ByteString, Any]`.

To set the request body as a stream:

```
import sttp.client._
import sttp.client.akkahttp._

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

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

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

To receive the response body as a stream:

```
import sttp.client._
import sttp.client.akkahttp._

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

implicit val sttpBackend = AkkaHttpBackend()

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

21.1 Testing

For testing, 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:

```
val backend: SttpBackend[Future, Nothing, Flow[Message, Message, *]] = {
  AkkaHttpBackend.usingClient(system, http = AkkaHttpClient.stubFromRoute(Routes.
↪route))
}
```

21.2 Websockets

The akka-http backend supports websockets, where the websocket handler is of type `akka.stream.scaladsl.Flow[Message, Message, _]`. That is, when opening a websocket connection, you need to provide the description of a stream, which will consume incoming websocket messages, and produce outgoing websocket messages. For example:

```
import akka.Done
import akka.stream.scaladsl.Flow
import akka.http.scaladsl.model.ws.Message

import sttp.client._
import sttp.client.ws.WebSocketResponse

import scala.concurrent.Future

val flow: Flow[Message, Message, Future[Done]] = ...
val response: Future[WebSocketResponse[Future[Done]]] =
    basicRequest.get(uri"wss://echo.websocket.org").openWebsocket(flow)
```

In this example, the given flow materialises to a `Future[Done]`, however this value can be arbitrary and depends on the shape and definition of the message-processing stream. The `Future[WebSocketResponse]` will complete once the websocket is established and contain the materialised value.

CHAPTER 22

async-http-client backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "async-http-client-backend-future" % "2.0.0-RC6"
// or
"com.softwaremill.sttp.client" %% "async-http-client-backend-scalaz" % "2.0.0-RC6"
// or
"com.softwaremill.sttp.client" %% "async-http-client-backend-zio" % "2.0.0-RC6"
// or
"com.softwaremill.sttp.client" %% "async-http-client-backend-zio-streams" % "2.0.0-RC6
↪"
// or
"com.softwaremill.sttp.client" %% "async-http-client-backend-monix" % "2.0.0-RC6"
// or
"com.softwaremill.sttp.client" %% "async-http-client-backend-cats" % "2.0.0-RC6"
// or
"com.softwaremill.sttp.client" %% "async-http-client-backend-fs2" % "2.0.0-RC6"
```

This backend depends on async-http-client. A fully **asynchronous** backend, which uses Netty behind the scenes.

The responses are wrapped depending on the dependency chosen in either a:

- standard Scala `Future`
- Scalaz `Task`. There's a transitive dependency on `scalaz-concurrent`.
- ZIO `IO`. There's a transitive dependency on `zio`.
- Monix `Task`. There's a transitive dependency on `monix-eval`.
- Any type implementing the Cats Effect `Async` typeclass, such as `cats.effect.IO`. There's a transitive dependency on `cats-effect`.
- fs2 `Stream`. There are transitive dependencies on `fs2`, `fs2-reactive-streams`, and `cats-effect`.

Next you'll need to add an implicit value:

```
implicit val sttpBackend = AsyncHttpClientFutureBackend()

// or, if you're using the scalaz version:
implicit val sttpBackend = AsyncHttpClientScalazBackend()

// or, if you're using the zio version:
implicit val sttpBackend = AsyncHttpClientZioBackend()

// or, if you're using the zio version with zio-streams for http streaming:
implicit val sttpBackend = AsyncHttpClientZioStreamsBackend()

// or, if you're using the monix version:
implicit val sttpBackend = AsyncHttpClientMonixBackend()

// or, if you're using the cats effect version:
implicit val sttpBackend = AsyncHttpClientCatsBackend[cats.effect.IO]()

// or, if you're using the fs2 version:
implicit val sttpBackend = AsyncHttpClientFs2Backend[cats.effect.IO]()

// or, if you'd like to use custom configuration:
implicit val sttpBackend = AsyncHttpClientFutureBackend.
↪usingConfig(asyncHttpClientConfig)

// or, if you'd like to use adjust the configuration sttp creates:
implicit val sttpBackend = AsyncHttpClientFutureBackend.
↪usingConfigBuilder(adjustFunction, sttpOptions)

// or, if you'd like to instantiate the AsyncHttpClient yourself:
implicit val sttpBackend = AsyncHttpClientFutureBackend.usingClient(asyncHttpClient)
```

22.1 Streaming using Monix

The Monix backend supports streaming (as both Monix and Async Http Client support reactive streams `Publisher`s out of the box). The type of supported streams in this case is `Observable[ByteBuffer]`. That is, you can set such an observable as a request body:

```
import sttp.client._
import sttp.client.asynchttpclient.monix._

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

AsyncHttpClientMonixBackend().flatMap { implicit backend =>
  val obs: Observable[ByteBuffer] =  ...

  basicRequest
    .streamBody(obs)
    .post(uri"...")
}
```

And receive responses as an observable stream:

```
import sttp.client._
import sttp.client.asynchttpclient.monix._
```

(continues on next page)

(continued from previous page)

```
import java.nio.ByteBuffer
import monix.eval.Task
import monix.reactive.Observable
import scala.concurrent.duration.Duration

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

22.2 Streaming using fs2

The fs2 backend supports streaming in 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, ByteBuffer]`.

Requests can be sent with a streaming body like this:

```
import sttp.client._
import sttp.client.asynchttpclient.fs2.AsyncHttpClientFs2Backend

import java.nio.ByteBuffer
import cats.effect.{ContextShift, IO}
import fs2.Stream

implicit val cs: ContextShift[IO] = IO.contextShift(ExecutionContext.Implicits.global)
val effect = AsyncHttpClientFs2Backend[IO]().flatMap { implicit backend =>
  val stream: Stream[IO, ByteBuffer] = ...

  basicRequest
    .streamBody(stream)
    .post(uri"...")
}
// run the effect
```

Responses can also be streamed:

```
import sttp.client._
import sttp.client.asynchttpclient.fs2.AsyncHttpClientFs2Backend

import java.nio.ByteBuffer
import cats.effect.{ContextShift, IO}
import fs2.Stream
import scala.concurrent.duration.Duration

implicit val cs: ContextShift[IO] = IO.contextShift(ExecutionContext.Implicits.global)
val effect = AsyncHttpClientFs2Backend[IO]().flatMap { implicit backend =>
  val response: IO[Response[Either[String, Stream[IO, ByteBuffer]]]] =
    basicRequest
      .post(uri"...")
```

(continues on next page)

(continued from previous page)

```
      .response(asStream[Stream[IO, ByteBuffer]])
      .readTimeout(Duration.Inf)
      .send()

  response
}
// run the effect
```

22.3 Websockets

The async-http-client backend supports websockets, where the websocket handler is of type `sttp.client.asynchttpclient.WebSocketHandler`. An instance of this handler can be created in two ways.

First (the "low-level" one), given an async-http-client-native `org.asynchttpclient.ws.WebSocketListener`, you can lift it to a web socket handler using `WebSocketHandler.fromListener`. This listener will receive lifecycle callbacks, as well as a callback each time a message is received. Note that the callbacks will be executed on the Netty (network) thread, so make sure not to run any blocking operations there, and delegate to other executors/thread pools if necessary. The value returned in the `WebSocketResponse` will be an instance of `org.asynchttpclient.ws.WebSocket`, which allows sending messages.

The second, "high-level" approach, available when using the Monix, ZIO and fs2 backends, is to pass a `MonixWebSocketHandler()`, `ZIOWebSocketHandler()` or `Fs2WebSocketHandler()`. This will create a websocket handler and expose a `sttp.client.ws.WebSocket[Task]` (for Monix and ZIO) / `sttp.client.ws.WebSocket[F]` (for fs2 and any `F[_] : ConcurrentEffect`) interface for sending/receiving messages.

See *websockets* for details on how to use the high-level interface.

22.4 Streaming websockets using fs2

For fs2, there are additionally some high-level helpers collected in `sttp.client.asynchttpclient.fs2.Fs2Websockets` which provide means to run the whole websocket communication through an `fs2.Pipe`. Example for a simple echo client:

```
import cats.effect.IO
import cats.implicits._
import sttp.client._
import sttp.client.ws._
import sttp.model.ws.WebSocketFrame

basicRequest
  .get(uri"wss://echo.websocket.org")
  .openWebsocketF(Fs2WebSocketHandler())
  .flatMap { response =>
    Fs2WebSockets.handleSocketThroughTextPipe(response.result) { in =>
      val receive = in.evalMap(m => IO(println("Received"))
      val send = Stream("Message 1".asRight, "Message 2".asRight, WebSocketFrame.
↪close.asLeft)
      send merge receive.drain
    }
  }
```

CHAPTER 23

OkHttp backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "okhttp-backend" % "2.0.0-RC6"
// or, for the monix version:
"com.softwaremill.sttp.client" %% "okhttp-backend-monix" % "2.0.0-RC6"
```

This backend depends on OkHttp, and offers:

- a **synchronous** backend: `OkHttpSyncBackend`
- an **asynchronous**, `Future`-based backend: `OkHttpFutureBackend`
- an **asynchronous**, Monix-`Task`-based backend: `OkHttpMonixBackend`

OkHttp fully supports HTTP/2.

23.1 Websockets

First (the "low-level" one), given an OkHttp-native `okhttp3.WebSocketListener`, you can lift it to a web socket handler using `WebSocketHandler.fromListener`. This listener will receive lifecycle callbacks, as well as a callback each time a message is received. Note that the callbacks will be executed on the Netty (network) thread, so make sure not to run any blocking operations there, and delegate to other executors/thread pools if necessary. The value returned in the `WebSocketResponse` will be an instance of `okhttp3.WebSocket`, which allows sending messages.

The second, "high-level" approach, available when using the Monix variant, is to pass a `MonixWebSocketHandler()`. This will create a websocket handler and expose a `sttp.client.ws.WebSocket[Task]` interface for sending/receiving messages.

See *websockets* for details on how to use the high-level interface.

CHAPTER 24

Http4s backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "http4s-backend" % "2.0.0-RC6"
```

This backend depends on http4s (blaze client), and offers an asynchronous backend, which can wrap results in any type implementing the cats-effect `Effect` typeclass.

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 the `RequestT.options.readTimeout` option

Instead, all custom timeout configuration should be done by creating a `org.http4s.client.Client[F]`, using `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 with the async-http-backend.

CHAPTER 25

Finagle backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "finagle-backend" % "2.0.0-RC6"
```

Next you'll need to add an implicit value:

```
import sttp.client.finagle.FinagleBackend
implicit val sttpBackend = 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 streaming or websockets

CHAPTER 26

HttpClient backend (Java 11+)

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "httpclient-backend" % "2.0.0-RC6"
```

This backend is based on `java.net.http.HttpClient` available from Java 11 onwards and offers:

- a **synchronous** backend: `HttpClientSyncBackend`
- an **asynchronous**, `Future`-based backend: `HttpClientFutureBackend`
- an **asynchronous**, `Monix-Task`-based backend: `HttpClientMonixBackend` with streaming support

26.1 Websockets

The HttpClient backend supports websockets, where the websocket handler is of type `sttp.client.httpclient.WebSocketHandler`. An instance of this handler can be created in two ways.

First (the "low-level" one), given an HttpClient-native `java.net.http.WebSocket.Listener`, you can lift it to a web socket handler using `WebSocketHandler.fromListener`. This listener will receive lifecycle callbacks, as well as a callback each time a message is received. Note that the callbacks will be executed on the Netty (network) thread, so make sure not to run any blocking operations there, and delegate to other executors/thread pools if necessary. The value returned in the `WebSocketResponse` will be an instance of `java.net.http.WebSocket`, which allows sending messages.

The second, "high-level" approach, available when using the Monix variant, is to pass a `MonixWebSocketHandler()`. This will create a websocket handler and expose a `sttp.client.ws.WebSocket[Task]` interface for sending/receiving messages.

See *websockets* for details on how to use the high-level interface.

CHAPTER 27

Opentracing backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "opentracing-backend" % "2.0.0-M7"
```

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.client.brave.OpenTracingBackend._

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

27.1 Integration with jaeger

Using with jaeger tracing

Add following dependency:

```
libraryDependencies += "io.jaegertracing" % "jaeger-client" % "1.0.0"
```

Create an instance of tracer:

```
import io.jaegertracing.Configuration
import io.jaegertracing.Configuration.ReporterConfiguration
import io.jaegertracing.Configuration.SamplerConfiguration
import io.jaegertracing.internal.JaegerTracer

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

27.2 Integration with brave

Using with brave tracing

Add following dependency:

```
libraryDependencies += "io.opentracing.brave" % "brave-opentracing" % "0.34.2"
```

Create instance of tracer:

```
def initTracer(zipkinUrl: String, serviceName: String): Tracer = {
    // Configure a reporter, which controls how often spans are sent
    //   (the dependency is io.zipkin.reporter2:zipkin-sender-okhttp3)
    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

CHAPTER 28

brave backend (deprecated)

Since 2.0.0-RC6 brave-backend is deprecated, you should use *opentracing backend* with brave integration.

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "brave-backend" % "2.0.0-RC6"
```

This backend depends on brave, a distributed tracing implementation compatible with Zipkin backend services.

The brave backend wraps any other backend, and needs an instance of brave's `HttpTracing` or `Tracing`, for example:

```
val httpTracing: HttpTracing = ...
implicit val sttpBackend = BraveBackend(AkkaHttpBackend(), httpTracing)
```

The backend obtains the current trace context using default Brave's propagation mechanisms. As it's often challenging to integrate context propagation in an asynchronous setting, there's also a possibility to add the trace context to the request's tags:

```
import sttp.client.brave.BraveBackend._

val parent: TraceContext = ...

basicRequest
  .get(...)
  .tagWithTraceContext(parent))
```

CHAPTER 29

Prometheus backend

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %% "prometheus-backend" % "2.0.0-RC6"
```

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:

```
implicit val sttpBackend = 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:

```
implicit val sttpBackend = PrometheusBackend(AkkaHttpBackend(), request =>␣
↪Some(request.uri.host))
```

You can disable request histograms by passing `None` returning function:

```
implicit val sttpBackend = PrometheusBackend(AkkaHttpBackend(), _ => 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:

```
implicit val sttpBackend = PrometheusBackend(AkkaHttpBackend(),␣
↪requestToInProgressGaugeNameMapper = request => Some(request.uri.host))
```

You can disable request in-progress gauges by passing `None` returning function:

```
implicit val sttpBackend = PrometheusBackend(AkkaHttpBackend(),␣
↪requestToInProgressGaugeNameMapper = _ => None)
```

CHAPTER 30

Fetch backend

A JavaScript backend implemented using the Fetch API and backed via `Future`.

To use, add the following dependency to your project:

```
"com.softwaremill.sttp.client" %%% "core" % "2.0.0-RC6"
```

And add an implicit value:

```
implicit val sttpBackend = 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.

30.1 Node.js

Running sttp in a node.js will require downloading modules that implement the various classes and functions used by sttp, usually available in browser. At minima, you will need replacement for `fetch`, `AbortController` and `Headers`. To achieve this, you can either use `npm` directly, or the `scalajs-bundler` sbt plugin if you use sbt

```
npm install --save node-fetch
npm install --save abortcontroller-polyfill
npm install --save fetch-headers
```

You then need to load the modules into your runtime. This can be done in your main method as such

```
val g = scalajs.js.Dynamic.global
g.fetch = g.require("node-fetch")
g.require("abortcontroller-polyfill/dist/polyfill-patch-fetch")
g.Headers = g.require("fetch-headers")
```

30.2 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.client" %%% "monix" % "2.0.0-RC6"
```

An example of streaming a response:

```
import sttp.client._
import sttp.client.impl.monix._

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

implicit val sttpBackend = FetchMonixBackend()

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

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.

CHAPTER 31

Custom backends, logging, metrics

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 response 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)

31.1 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.

31.2 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:

```
class MyWrapper[F[_], S, WS_HANDLER[_]] private (delegate: SttpBackend[F, S, WS_
↪HANDLER])
  extends SttpBackend[R, S, WS_HANDLER] {

  ...
}

object MyWrapper {
  def apply[F[_], S, WS_HANDLER[_]](
    delegate: SttpBackend[F, S, WS_HANDLER]): SttpBackend[F, S, WS_HANDLER] = {
    // disables any other FollowRedirectsBackend-s further down the delegate chain
    new FollowRedirectsBackend(new MyWrapper(delegate))
  }
}
```

31.3 Example logging backend wrapper

Often it's useful to setup system-wide logging for failed requests. This is possible using a backend wrapper. In this example, we are using `scala-logging` for the logging itself, but of course any logging library can be used:

```
import sttp.client.{MonadError, Request, Response, SttpBackend}
import com.typesafe.scalalogging.StrictLogging

class LoggingSttpBackend[F[_], S, WS_HANDLER[_]](delegate: SttpBackend[R, S, WS_
↪HANDLER])
  extends SttpBackend[R, S, WS_HANDLER]
  with StrictLogging {

  override def send[T](request: Request[T, S]): F[Response[T]] = {
    responseMonad.map(responseMonad.handleError(delegate.send(request)) {
      case e: Exception =>
        logger.error(s"Exception when sending request: $request", e)
        responseMonad.error(e)
    }) { response =>
      if (response.isSuccess) {
        logger.debug(s"For request: $request got response: $response")
      } else {
        logger.warn(s"For request: $request got response: $response")
      }
      response
    }
  }
  override def openWebsocket[T, WS_RESULT](
      request: Request[T, S],
      handler: WS_HANDLER[WS_RESULT]
    ): F[WebSocketResponse[WS_RESULT]] = {
    responseMonad.map(responseMonad.handleError(delegate.openWebsocket(request,␣
↪handler)) {
      case e: Exception =>
        logger.error(s"Exception when opening a websocket request: $request", e)
        responseMonad.error(e)
```

(continues on next page)

(continued from previous page)

```
      }) { response =>
        logger.debug(s"For ws request: $request got headers: ${response.headers}")
        response
      }
  }
  override def close(): F[Unit] = delegate.close()
  override def responseMonad: MonadError[F] = delegate.responseMonad
}
```

Note that there are three possible outcomes of a request:

- an exception is thrown (handled with `responseMonad.handleError`), e.g. because of a connection error; here, this is logged with level `ERROR`.
- the response completes normally, but the server returns a non-2xx response code. Here, this case is logged with level `WARN`.
- the response completes normally with 2xx response code. Here, this case is logged with level `DEBUG`.

It's quite easy to customize this backend to your particular needs - just copy the code!

31.4 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:

```
// 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[S](delegate: SttpBackend[Future, S, NothingT],
                       metrics: MetricsServer)
    extends SttpBackend[Future, S, NothingT] {

  override def send[T](request: Request[T, S]): 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)

```
  override def openWebsocket[T, WS_RESULT](
      request: Request[T, S],
      handler: NothingT[WS_RESULT]
    ): Future[WebSocketResponse[WS_RESULT]] = {
    delegate.openWebsocket(request, handler) // No websocket support due to NothingT
  }

  override def close(): F[Unit] = delegate.close()

  override def responseMonad: MonadError[Future] = delegate.responseMonad
}

// example usage
implicit val backend = new MetricWrapper(
  AkkaHttpBackend(),
  new CloudMetricsServer()
)

basicRequest
  .get(uri"http://company.com/api/service1")
  .tag("metric", "service1")
  .send()
```

31.5 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.client.{MonadError, Request, Response, SttpBackend}

class RetryingBackend[F[_], S](
    delegate: SttpBackend[F, S, NothingT],
    shouldRetry: (Request[_, _], Either[Throwable, Response[_]]) => Boolean,
    maxRetries: Int)
    extends SttpBackend[F, S, NothingT] {

  override def send[T](request: Request[T, S]): F[Response[T]] = {
    sendWithRetryCounter(request, 0)
  }

  private def sendWithRetryCounter[T](request: Request[T, S],
                                      retries: Int): F[Response[T]] = {
    val r = responseMonad.handleError(delegate.send(request)) {
```

(continues on next page)

(continued from previous page)

```
      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)
      } else {
        responseMonad.unit(resp)
      }
    }
  }

  override def openWebsocket[T, WS_RESULT](
      request: Request[T, S],
      handler: NothingT[WS_RESULT]
    ): Future[WebSocketResponse[WS_RESULT]] = {
    delegate.openWebsocket(request, handler) // No websocket support due to NothingT
  }

  override def close(): F[Unit] = delegate.close()

  override def responseMonad: MonadError[F] = delegate.responseMonad
}
```

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

31.6 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.client.monad.MonadError
import sttp.client.ws.WebSocketResponse
import sttp.client.{Request, Response, SttpBackend}
import java.util.concurrent.TimeUnit

class CircuitSttpBackend[F[_], S, W[_]](
    circuitBreaker: CircuitBreaker,
    delegate: SttpBackend[F, S, W]
    )(implicit monadError: MonadError[F]) extends SttpBackend[F, S, W] {

  override def send[T](request: Request[T, S]): F[Response[T]] = {
    CircuitSttpBackend.decorateF(circuitBreaker, delegate.send(request))
  }
```

(continues on next page)

(continued from previous page)

```
  override def openWebsocket[T, WS_RESULT](
      request: Request[T, S],
      handler: W[WS_RESULT]
  ): F[WebSocketResponse[WS_RESULT]] =
        CircuitSttpBackend.decorateF(delegate.openWebsocket(request, handler))

  override def close(): F[Unit] = delegate.close()

  override def responseMonad: MonadError[F] = delegate.responseMonad
}

object CircuitSttpBackend {

  def decorateF[F[_], T](
      circuitBreaker: CircuitBreaker,
      service: => F[T]
  )(implicit monadError: MonadError[F]): F[T] = {

    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)
            monadError.error(t)
        }
      } catch {
        case t: Throwable =>
          circuitBreaker.onError(System.nanoTime() - start, TimeUnit.NANOSECONDS, t)
          monadError.error(t)
      }
    }
  }
}
```

31.7 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.client.monad.MonadError
import sttp.client.ws.WebSocketResponse
import sttp.client.{Request, Response, SttpBackend}

class RateLimitingSttpBackend[F[_], S, W[_]](
```

(continues on next page)

(continued from previous page)

```
    rateLimiter: RateLimiter,
    delegate: SttpBackend[F, S, W]
    )(implicit monadError: MonadError[F]) extends SttpBackend[F, S, W] {

  override def send[T](request: Request[T, S]): F[Response[T]] = {
    RateLimitingSttpBackend.decorateF(rateLimiter, delegate.send(request))
  }

  override def openWebsocket[T, WS_RESULT](
      request: Request[T, S],
      handler: W[WS_RESULT]
  ): F[WebSocketResponse[WS_RESULT]] = delegate.openWebsocket(request, handler)

  override def close(): F[Unit] = delegate.close()

  override def responseMonad: MonadError[F] = delegate.responseMonad
}

object RateLimitingSttpBackend {

  def decorateF[F[_], T](
      rateLimiter: RateLimiter,
      service: => F[T]
  )(implicit monadError: MonadError[F]): F[T] = {
    try {
      RateLimiter.waitForPermission(rateLimiter)
      service
    } catch {
      case t: Throwable =>
        monadError.error(t)
    }
  }
}
```

31.8 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.client" %% "core" % "2.0.0-RC6" % "test" classifier "tests",
"com.typesafe.akka" %% "akka-http" % "10.1.1" % "test",
"ch.megard" %% "akka-http-cors" % "0.3.0" % "test",
"com.typesafe.akka" %% "akka-stream" % "2.5.12" % "test",
"org.scalatest" %% "scalatest" % "3.0.5" % "test"
```

Implement your backend and extend the `HttpTest` class:

```
import sttp.client.SttpBackend
import sttp.client.testing.{ConvertToFuture, HttpTest}

class MyCustomBackendHttpTest extends HttpTest[Future] {

  override implicit val convertToFuture: ConvertToFuture[Future] = ConvertToFuture.
↪future
```

(continues on next page)

(continued from previous page)

```
  override implicit lazy val backend: SttpBackend[Future, Nothing, NothingT] = new␣
↪MyCustomBackend()

}
```

You can find a more detailed example in the sttp-vertx repository.

CHAPTER 32

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*.

32.1 Creating a stub backend

An empty backend stub can be created using the following ways:

- by using one of the factory methods `SttpBackendStub.synchronous` or `SttpBackendStub.asynchronousFuture`, which return stubs which use the `Id` or standard Scala's `Future` response wrappers without streaming support
- by explicitly giving the response wrapper monad and supported streams type, e.g. `SttpBackendStub[Task, Observable[ByteBuffer]](TaskMonad)`
- given an instance of a "real" backend, e.g. `SttpBackendStub(HttpURLConnectionBackend())` or `SttpBackendStub(AsyncHttpClientScalazBackend())`. The stub will then use the same response wrapper and support the same type of streams as the given "real" backend.
- by specifying a fallback/delegate backend, see below

32.2 Specifying behavior

Behavior of the stub can be specified using a combination of the `whenRequestMatches` and `thenRespond` methods:

```
implicit val testingBackend = SttpBackendStub.synchronous
  .whenRequestMatches(_.uri.path.startsWith(List("a", "b")))
  .thenRespond("Hello there!")
```

(continues on next page)

(continued from previous page)

```
  .whenRequestMatches(_.method == Method.POST)
  .thenRespondServerError()

val response1 = basicRequest.get(uri"http://example.org/a/b/c").send()
// response1.body will be Right("Hello there")

val response2 = basicRequest.post(uri"http://example.org/d/e").send()
// response2.code will be 500
```

It is also possible to match requests by partial function, returning a response. E.g.:

```
implicit val testingBackend = SttpBackendStub.synchronous
  .whenRequestMatchesPartial({
    case r if r.uri.path.endsWith(List("partial10")) =>
      Response.error("Not found", 404)

    case r if r.uri.path.endsWith(List("partialAda")) =>
      // additional verification of the request is possible
      assert(r.body == StringBody("z"))
      Response.ok("Ada")
  })

val response1 = basicRequest.get(uri"http://example.org/partial10").send()
// response1.body will be Right(10)

val response2 = basicRequest.post(uri"http://example.org/partialAda").send()
// response2.body will be Right("Ada")
```

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 result monad 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:

```
implicit val testingBackend = SttpBackendStub.asynchronousFuture.whenAnyRequest
  .thenRespondWrapped(Future {
    Thread.sleep(5000)
    Response(Right("OK"), 200, "", Nil, Nil)
  })

val responseFuture = basicRequest.get(uri"http://example.org").send()
// responseFuture will complete after 5 seconds with "OK" response
```

The returned response may also depend on the request:

```
implicit val testingBackend = SttpBackendStub.synchronous.whenAnyRequest
  .thenRespondWrapped(req =>
    Response(Right("OK, got request sent to ${req.uri.host}"), 200, "", Nil, Nil)
  )

val response = basicRequest.get(uri"http://example.org").send()
// response.body will be Right("OK, got request sent to example.org")
```

You can define consecutive raw responses that will be served:

```scala
implicit val testingBackend = SttpBackendStub.synchronous.whenAnyRequest
  .thenRespondCyclic("first", "second", "third")

basicRequest.get(uri"http://example.org").send()       // Right("OK, first")
basicRequest.get(uri"http://example.org").send()       // Right("OK, second")
basicRequest.get(uri"http://example.org").send()       // Right("OK, third")
basicRequest.get(uri"http://example.org").send()       // Right("OK, first")
```

Or multiple *Response* instances:

```scala
implicit val testingBackend = SttpBackendStub.synchronous.whenAnyRequest
  .thenRespondCyclicResponses(
    Response.ok[String]("first"),
    Response.error[String]("error", 500, "Something went wrong")
  )

basicRequest.get(uri"http://example.org").send()       // code will be 200
basicRequest.get(uri"http://example.org").send()       // code will be 500
basicRequest.get(uri"http://example.org").send()       // code will be 200
```

32.3 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.:

```scala
implicit val testingBackend = SttpBackendStub.synchronous
  .whenRequestMatches(_ => true)
  .thenRespond(throw new TimeoutException())
```

32.4 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`

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]`
- `InputStream`, `String` and `Array[Byte]` to custom types through mapped response specifications

32.5 Example: returning JSON

For example, if you want to return a JSON response, simply use *.withResponse(String)* as below::

```
implicit 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(asByteArray.map(parseUserJson))
  .send()
```

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.

Note that no conversions will be attempted for streaming response bodies.

32.6 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 mock 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:

```
val sourceFile = new File("path/to/file.ext")
val destinationFile = new File("path/to/file.ext")
SttpBackendStub.synchronous
  .whenRequestMatches(_ => true)
  .thenRespondWrapped { _ =>
    FileUtils.copyFile(sourceFile, destinationFile) // org.apache.commons.io
    IO(Response(Right(destinationFile, 200, ""))
  }
```

32.7 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:

```scala
implicit 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()
// response1.body will be Right("I'm a STUB")

val response2 = basicRequest.post(uri"http://api.internal/b").send()
// response2 will be whatever a "real" network call to api.internal/b returns
```

CHAPTER 33

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.client._
import scala.concurrent.duration._

// all backends provide a constructor that allows to specify backend options
implicit val backend = HttpURLConnectionBackend(
  options = SttpBackendOptions.connectionTimeout(1.minute))

sttp
  .get(uri"...")
  .readTimeout(5.minutes) // or Duration.Inf to turn read timeout off
  .send()
```

CHAPTER 34

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

CHAPTER 35

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)* 2. `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.client._

implicit val backend = HttpURLConnectionBackend(
  options = SttpBackendOptions.httpProxy("some.host", 8080))

sttp
  .get(uri"...")
  .send() // uses the proxy
```

Or in case your proxy requires authentication (supported by the JVM backends):

```
SttpBackendOptions.httpProxy("some.host", 8080, "username", "password")
```

CHAPTER 36

Redirects

By default, sttp follows redirects.

If you'd like to disable following redirects, use the `followRedirects` method:

```
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.

36.1 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:

```
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.

CHAPTER 37

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.

Also read about handling non 2xx responses if you need to unmarshal error responses.

37.1 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.client" %% "circe" % "2.0.0-RC6"
```

This module adds a method to the request and a function that can be given to a request to decode the response to a specific object:

```
import sttp.client._
import sttp.client.circe._

implicit val backend = HttpURLConnectionBackend()

// Assume that there is an implicit circe encoder in scope
// for the request Payload, and a decoder for the MyResponse
val requestPayload: Payload = ???

val response: Response[Either[ResponseError[io.circe.Error], MyResponse]] =
  basicRequest
    .post(uri"...")
    .body(requestPayload)
    .response(asJson[MyResponse])
    .send()
```

37.2 Json4s

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

```
"com.softwaremill.sttp.client" %% "json4s" % "2.0.0-RC6"
"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.client._
import sttp.client.json4s._

implicit val backend = HttpURLConnectionBackend()

case class Payload(...)
case class MyResponse(...)

val requestPayload: Payload = Payload(...)

implicit val serialization = org.json4s.native.Serialization

val response: Response[Either[ResponseError[Exception], MyResponse], Nothing] =
  basicRequest
    .post(uri"...")
    .body(requestPayload)
    .response(asJson[MyResponse])
    .send()
```

37.3 spray-json

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

```
"com.softwaremill.sttp.client" %% "spray-json" % "2.0.0-RC6"
```

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.client._
import sttp.client.sprayJson._
import spray.json._

implicit val backend = HttpURLConnectionBackend()

case class Payload(...)
object Payload {
  implicit val jsonFormat: RootJsonFormat[Payload] = ...
```

(continues on next page)

(continued from previous page)

```
}

case class MyResponse(...)
object MyResponse {
  implicit val jsonFormat: RootJsonFormat[MyResponse] = ...
}

val requestPayload: Payload = Payload(...)

val response: Response[Either[ResponseError[io.circe.Error], MyResponse]] =
  basicRequest
    .post(uri"...")
    .body(requestPayload)
    .response(asJson[MyResponse])
    .send()
```

37.4 play-json

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

```
"com.softwaremill.sttp.client" %% "play-json" % "2.0.0-RC6"
```

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

CHAPTER 38

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.

CHAPTER 39

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)