Package ‘crul’
February 15, 2021

Title HTTP Client

Description A simple HTTP client, with tools for making HTTP requests, and mocking HTTP requests. The package is built on R6, and takes inspiration from Ruby’s ‘faraday’ gem (<https://rubygems.org/gems/faraday>). The package name is a play on curl, the widely used command line tool for HTTP, and this package is built on top of the R package ‘curl’, an interface to ‘libcurl’ (<https://curl.se/libcurl/>).

Version 1.1.0

License MIT + file LICENSE

URL https://docs.ropensci.org/crul/ (website)
https://github.com/ropensci/crul (devel)
https://books.ropensci.org/http-testing/ (user manual)

BugReports https://github.com/ropensci/crul/issues

Encoding UTF-8

Language en-US

Imports curl (>= 3.3), R6 (>= 2.2.0), urltools (>= 1.6.0), httpcode (>= 0.2.0), jsonlite, mime

Suggests testthat, roxygen2 (>= 7.1.1), fauxpas (>= 0.1.0), webmockr (>= 0.1.0), knitr, rmarkdown

VignetteBuilder knitr

RoxygenNote 7.1.1

X-schema.org-applicationCategory Web

X-schema.org-keywords http, https, API, web-services, curl, download, libcurl, async, mocking, caching

X-schema.org-isPartOf https://ropensci.org

NeedsCompilation no

Author Scott Chamberlain [aut, cre] (<https://orcid.org/0000-0003-1444-9135>)

Maintainer Scott Chamberlain <myrmecocystus@gmail.com>

Repository CRAN

Date/Publication 2021-02-15 16:10:10 UTC
R topics documented:

- crul-package ................................................................. 2
- Async .......................................................... 4
- AsyncQueue .................................................. 11
- AsyncVaried .................................................. 13
- auth .......................................................... 18
- content-types .............................................. 19
- cookies ...................................................... 21
- crul-options .................................................. 22
- curl-options .................................................. 24
- curl_verbose .................................................. 25
- handle ......................................................... 26
- hooks ......................................................... 26
- http-headers ................................................. 27
- HttpClient .................................................. 28
- HttpRequest ................................................. 38
- HttpResponse .............................................. 43
- mock .......................................................... 47
- ok .............................................................. 48
- Paginator ..................................................... 50
- progress ..................................................... 57
- proxies ....................................................... 57
- upload ......................................................... 59
- url_build ..................................................... 59
- verb-DELETE .................................................. 60
- verb-GET ..................................................... 61
- verb-HEAD .................................................. 62
- verb-PATCH .................................................. 63
- verb-POST .................................................. 64
- verb-PUT ..................................................... 65
- writing-options ........................................... 66

Index ............................................................. 68

<table>
<thead>
<tr>
<th>crul-package</th>
<th>crul</th>
</tr>
</thead>
</table>

Description

HTTP R client
Package API

- **HttpClient()** - create a connection client, set all your http options, make http requests
- **HttpResponse()** - mostly for internal use, handles http responses
- **Paginator()** - auto-paginate through requests
- **Async()** - asynchronous requests
- **AsyncVaried()** - varied asynchronous requests
- **HttpRequest()** - generate an HTTP request, mostly for use in building requests to be used in Async or AsyncVaried
- **mock()** - Turn on/off mocking, via webmockr
- **auth()** - Simple authentication helper
- **proxy()** - Proxy helper
- **upload()** - File upload helper

- set curl options globally: `set_auth()`, `set_headers()`, `set_opts()`, `set_proxy()`, and `crul_settings()`

**HTTP verbs (or HTTP request methods)**

See verb-GET, verb-POST, verb-PUT, verb-PATCH, verb-DELETE, verb-HEAD for details.

- **HttpClient** is the main interface for making HTTP requests, and includes methods for each HTTP verb
- **HttpRequest** allows you to prepare a HTTP payload for use with AsyncVaried, which provides asynchronous requests for varied HTTP methods
- **Async** provides asynchronous requests for a single HTTP method at a time
- the verb() method can be used on all the above to request a specific HTTP verb

**Checking HTTP responses**

**HttpResponse()** has helpers for checking and raising warnings/errors.

- **content-types** details the various options for checking content types and throwing a warning or error if the response content type doesn’t match what you expect. Mis-matched content-types are typically a good sign of a bad response. There’s methods built in for json, xml and html, with the ability to set any custom content type
- **raise_for_status()** is a method on **HttpResponse()** that checks the HTTP status code, and errors with the appropriate message for the HTTP status code, optionally using the package fauxpas if it’s installed.

**HTTP conditions**

We use fauxpas if you have it installed for handling HTTP conditions but if it’s not installed we use **httpcode**
Async

Mocking

Mocking HTTP requests is supported via the webmockr package. See mock for guidance, and https://books.ropensci.org/http-testing/.

Caching

Caching HTTP requests is supported via the vcr package. See https://books.ropensci.org/http-testing/.

Links

Source code: https://github.com/ropensci/crul
Bug reports/feature requests: https://github.com/ropensci/crul/issues

Author(s)

Scott Chamberlain <myrmecocystus@gmail.com>

Async

Simple async client

Description

An async client to work with many URLs, but all with the same HTTP method

Details

See HttpClient() for information on parameters.

Value

a list, with objects of class HttpResponse(). Responses are returned in the order they are passed in. We print the first 10.

Failure behavior

HTTP requests mostly fail in ways that you are probably familiar with, including when there’s a 400 response (the URL not found), and when the server made a mistake (a 500 series HTTP status code).

But requests can fail sometimes where there is no HTTP status code, and no agreed upon way to handle it other than to just fail immediately.

When a request fails when using synchronous requests (see HttpClient) you get an error message that stops your code progression immediately saying for example:

- "Could not resolve host: https://foo.com"
- "Failed to connect to foo.com"
• "Resolving timed out after 10 milliseconds"

However, for async requests we don’t want to fail immediately because that would stop the subse-
quent requests from occurring. Thus, when we find that a request fails for one of the reasons above
we give back a HttpResponse object just like any other response, and:

• capture the error message and put it in the content slot of the response object (thus calls to
content and parse() work correctly)
• give back a 0 HTTP status code. we handle this specially when testing whether the request
was successful or not with e.g., the success() method

Public fields

urls (character) one or more URLs
opts any curl options
proxies named list of headers
auth an object of class auth
headers named list of headers

Methods

Public methods:

• Async$print()
• Async$new()
• Async$get()
• Async$post()
• Async.put()
• Async$patch()
• Async$delete()
• Async$head()
• Async$verb()
• Async$clone()

Method print(): print method for Async objects

Usage:
Async$print(x, ...)

Arguments:
x self
... ignored

Method new(): Create a new Async object

Usage:
Async$new(urls, opts, proxies, auth, headers)

Arguments:
urls (character) one or more URLs
opts any curl options
proxies a proxy() object
auth an auth() object
headers named list of headers

Returns: A new Async object.

Method get(): execute the GET http verb for the urls

Usage:
Async$get(path = NULL, query = list(), disk = NULL, stream = NULL, ...)

Arguments:
path (character) URL path, appended to the base URL
query (list) query terms, as a named list
disk a path to write to. if NULL (default), memory used. See curl::curl_fetch_disk() for help.
stream an R function to determine how to stream data. if NULL (default), memory used. See curl::curl_fetch_stream() for help
... curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Examples:
\dontrun{
  (cc <- Async$new(urls = c(
    'https://httpbin.org/',
    'https://httpbin.org/get?a=5',
    'https://httpbin.org/get?foo=bar'
  )))
  (res <- cc$get())
}

Method post(): execute the POST http verb for the urls

Usage:
Async$post(
  path = NULL,
  query = list(),
  body = NULL,
  encode = "multipart",
  disk = NULL,
  stream = NULL,
  ...)

Arguments:
path (character) URL path, appended to the base URL
query (list) query terms, as a named list
body body as an R list
encode  one of form, multipart, json, or raw
disk  a path to write to. if NULL (default), memory used. See curl::curl_fetch_disk() for help.
stream an R function to determine how to stream data. if NULL (default), memory used. See curl::curl_fetch_stream() for help
... curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method put(): execute the PUT http verb for the urls

Usage:
Async$put(
  path = NULL,
  query = list(),
  body = NULL,
  encode = "multipart",
  disk = NULL,
  stream = NULL,
  ...
)

Arguments:
path  (character) URL path, appended to the base URL
query  (list) query terms, as a named list
body  body as an R list
encode  one of form, multipart, json, or raw
disk  a path to write to. if NULL (default), memory used. See curl::curl_fetch_disk() for help.
stream an R function to determine how to stream data. if NULL (default), memory used. See curl::curl_fetch_stream() for help
... curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method patch(): execute the PATCH http verb for the urls

Usage:
Async$patch(
  path = NULL,
  query = list(),
  body = NULL,
  encode = "multipart",
  disk = NULL,
  stream = NULL,
  ...
)

Arguments:
path  (character) URL path, appended to the base URL
query  (list) query terms, as a named list
body  body as an R list
encode  one of form, multipart, json, or raw
disk  a path to write to. if NULL (default), memory used. See `curl::curl_fetch_disk()` for help.
stream  an R function to determine how to stream data. if NULL (default), memory used. See `curl::curl_fetch_stream()` for help
...
  curl options, only those in the acceptable set from `curl::curl_options()` except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method delete(): execute the DELETE http verb for the urls

Usage:
Async$delete(
  path = NULL,
  query = list(),
  body = NULL,
  encode = "multipart",
  disk = NULL,
  stream = NULL,
  ...
)

Arguments:
path  (character) URL path, appended to the base URL
query  (list) query terms, as a named list
body  body as an R list
encode  one of form, multipart, json, or raw
disk  a path to write to. if NULL (default), memory used. See `curl::curl_fetch_disk()` for help.
stream  an R function to determine how to stream data. if NULL (default), memory used. See `curl::curl_fetch_stream()` for help
...
  curl options, only those in the acceptable set from `curl::curl_options()` except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method head(): execute the HEAD http verb for the urls

Usage:
Async$head(path = NULL, ...)

Arguments:
path  (character) URL path, appended to the base URL
...
  curl options, only those in the acceptable set from `curl::curl_options()` except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method verb(): execute any supported HTTP verb

Usage:
Async$verb(verb, ...)

Arguments:
verb (character) a supported HTTP verb: get, post, put, patch, delete, head.

... curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Examples:
\dontrun{
  cc <- Async$new(
    urls = c(
      'https://httpbin.org/',
      'https://httpbin.org/get?a=5',
      'https://httpbin.org/get?foo=bar'
    )
  )
  (res <- cc$verb('get'))
  lapply(res, function(z) z$parse("UTF-8"))
}

Method clone(): The objects of this class are cloneable with this method.

Usage:
Async$clone(deep = FALSE)

Arguments:
  deep  Whether to make a deep clone.

See Also

Other async: AsyncQueue, AsyncVaried, HttpRequest

Examples

## Not run:
cc <- Async$new(
  urls = c(
    'https://httpbin.org/',
    'https://httpbin.org/get?a=5',
    'https://httpbin.org/get?foo=bar'
  )
)
cc
(res <- cc$get())
res[[1]]$url
res[[1]]$success()
res[[1]]$status_http()
res[[1]]$response_headers
res[[1]]$method
res[[1]]$content
res[[1]]$parse("UTF-8")
lapply(res, function(z) z$parse("UTF-8"))

# curl options/headers with async
urls = c(
   'https://httpbin.org/',
   'https://httpbin.org/get?a=5',
   'https://httpbin.org/get?foo=bar'
)
c <- Async$new(urls = urls,
   opts = list(verbose = TRUE),
   headers = list(foo = "bar")
)
c
(res <- c$get())

# using auth with async
d <- Async$new(
   urls = rep('https://httpbin.org/basic-auth/user/passwd', 3),
   auth = auth(user = "Foo", pwd = "passwd"),
   opts = list(verbose = TRUE)
)
d
(res <- d$get())

vapply(res, function(z) z$status_code, double(1))
vapply(res, function(z) z$success(), logical(1))
lapply(res, function(z) z$parse("UTF-8"))

# failure behavior
## e.g. when a URL doesn't exist, a timeout, etc.
          "https://httpbin.org/get")
conn <- Async$new(urls = urls)
(res <- conn$get())
res[[1]]$parse("UTF-8") # a failure
res[[2]]$parse("UTF-8") # a failure
res[[3]]$parse("UTF-8") # a success

## End(Not run)

## Method Async$get

## Not run:
(cc <- Async$new(urls = c(
   'https://httpbin.org/',
   'https://httpbin.org/get?a=5',
   'https://httpbin.org/get?foo=bar'
)))
(res <- cc$get())

## End(Not run)

## Method Async$verb
AsyncQueue

## Not run:

```r
c <- Async$new(
  urls = c(
    'https://httpbin.org/=',
    'https://httpbin.org/get?a=5',
    'https://httpbin.org/get?foo=bar'
  )
)
res <- cc$verb('get')
lapply(res, function(z) z$parse("UTF-8"))
```

## End(Not run)

---

### Description

An AsyncQueue client

### Super class

crl::AsyncVaried -> AsyncQueue

### Public fields

- `bucket_size` (integer) number of requests to send at once
- `sleep` (integer) number of seconds to sleep between each bucket
- `req_per_min` (integer) requests per minute

### Methods

#### Public methods:

- `AsyncQueue$print()`
- `AsyncQueue$new()`
- `AsyncQueue$request()`
- `AsyncQueue$responses()`
- `AsyncQueue$clone()`

#### Method `print()`: print method for AsyncQueue objects

**Usage:**

```
AsyncQueue$print(x, ...)
```

**Arguments:**

- `x` self
... ignored

**Method new()**: Create a new AsyncQueue object

*Usage:*
```
AsyncQueue$new(
  ..., .list = list(),
  bucket_size = 5,
  sleep = NULL,
  req_per_min = NULL
)
```

*Arguments:*
`...`, `.list` Any number of objects of class `HttpRequest()`, must supply inputs to one of these parameters, but not both
`bucket_size` (integer) number of requests to send at once. default: 5. See Details.
`sleep` (integer) seconds to sleep between buckets. default: NULL (not set)
`req_per_min` (integer) maximum number of requests per minute. if NULL (default), its ignored

*Details:* Must set either `sleep` or `req_per_min`. If you set `req_per_min` we calculate a new `bucket_size` when `$new()` is called

*Returns:* A new AsyncQueue object

**Method request()**: Execute asynchronous requests

*Usage:*
```
AsyncQueue$request()
```

*Returns:* nothing, responses stored inside object, though will print messages if you choose verbose output

**Method responses()**: List responses

*Usage:*
```
AsyncQueue$responses()
```

*Returns:* a list of `HttpResponse` objects, empty list before requests made

**Method clone()**: The objects of this class are cloneable with this method.

*Usage:*
```
AsyncQueue$clone(deep = FALSE)
```

*Arguments:*
`deep` Whether to make a deep clone.

**See Also**

Other async: `AsyncVaried`, `Async`, `HttpRequest`
## Examples

```r
Not run:

# Using sleep
reqlist <- list(
  HttpRequest$new(url = "https://httpbin.org/get")$get(),
  HttpRequest$new(url = "https://httpbin.org/post")$post(),
  HttpRequest$new(url = "https://httpbin.org/put")$put(),
  HttpRequest$new(url = "https://httpbin.org/delete")$delete(),
  HttpRequest$new(url = "https://httpbin.org/get?g=5")$get(),
  HttpRequest$new(
    url = "https://httpbin.org/post")$post(body = list(y = 9)),
  HttpRequest$new(
    url = "https://httpbin.org/get")$get(query = list(hello = "world")),
  HttpRequest$new(url = "https://ropensci.org")$get(),
  HttpRequest$new(url = "https://ropensci.org/about")$get(),
  HttpRequest$new(url = "https://ropensci.org/packages")$get(),
  HttpRequest$new(url = "https://ropensci.org/community")$get(),
  HttpRequest$new(url = "https://ropensci.org/blog")$get(),
  HttpRequest$new(url = "https://ropensci.org/careers")$get()
)
out <- AsyncQueue$new(.list = reqlist, bucket_size = 5, sleep = 3)
out
out$bucket_size # bucket size
out$requests() # list requests
out$request() # make requests
out$responses() # list responses

# Using requests per minute
if (interactive()) {
  x="https://raw.githubusercontent.com/ropensci/roregistry/gh-pages/registry_urls.json"
  z <- HttpClient$new(x)$get()
  urls <- jsonlite::fromJSON(z$parse("UTF-8"))$git_url
  repos = Filter(length, regmatches(urls, gregexpr("ropensci/[A-Za-z]+", urls)))
  repos = unlist(repos)
  auth <- list(Authorization = paste("token", Sys.getenv("GITHUB_PAT")))
  reqs <- lapply(repos[1:50], function(w) {
    HttpRequest$new(paste0("https://api.github.com/repos/", w), headers = auth)$get()
  })
  out <- AsyncQueue$new(.list = reqs, req_per_min = 30)
  out
  out$bucket_size
  out$requests()
  out$request()
  out$responses()
}
## End(Not run)
```
Description

An async client to do many requests, each with different URLs, curl options, etc.

Value

An object of class AsyncVaried with variables and methods. HttpResponse objects are returned in the order they are passed in. We print the first 10.

Failure behavior

HTTP requests mostly fail in ways that you are probably familiar with, including when there’s a 400 response (the URL not found), and when the server made a mistake (a 500 series HTTP status code).

But requests can fail sometimes where there is no HTTP status code, and no agreed upon way to handle it other than to just fail immediately.

When a request fails when using synchronous requests (see HttpClient) you get an error message that stops your code progression immediately saying for example:

- "Could not resolve host: https://foo.com"
- "Failed to connect to foo.com"
- "Resolving timed out after 10 milliseconds"

However, for async requests we don’t want to fail immediately because that would stop the subsequent requests from occurring. Thus, when we find that a request fails for one of the reasons above we give back a HttpResponse object just like any other response, and:

- capture the error message and put it in the content slot of the response object (thus calls to content and parse() work correctly)
- give back a 0 HTTP status code. we handle this specially when testing whether the request was successful or not with e.g., the success() method

Methods

Public methods:

- AsyncVaried$print()
- AsyncVaried$new()
- AsyncVaried$request()
- AsyncVaried$responses()
- AsyncVaried$requests()
- AsyncVaried$parse()
- AsyncVaried$status_code()
- AsyncVaried$status()
- AsyncVaried$content()
- AsyncVaried$times()
- AsyncVaried$clone()

Method print(): print method for AsyncVaried objects
Usage:
AsyncVaried$print(x, ...)

Arguments:
  x  self
  ... ignored

Method new(): Create a new AsyncVaried object

Usage:
AsyncVaried$new(..., .list = list())

Arguments:
  ..., .list  Any number of objects of class HttpRequest(), must supply inputs to one of these
              parameters, but not both

Returns:  A new AsyncVaried object

Method request(): Execute asynchronous requests

Usage:
AsyncVaried$request()

Returns:  nothing, responses stored inside object, though will print messages if you choose
          verbose output

Method responses(): List responses

Usage:
AsyncVaried$responses()

Returns:  a list of HttpResponse objects, empty list before requests made

Method requests(): List requests

Usage:
AsyncVaried$requests()

Returns:  a list of HttpRequest objects, empty list before requests made

Method parse(): parse content

Usage:
AsyncVaried$parse(encoding = "UTF-8")

Arguments:
  encoding  (character) the encoding to use in parsing. default:"UTF-8"

Returns:  character vector, empty character vector before requests made

Method status_code(): Get HTTP status codes for each response

Usage:
AsyncVaried$status_code()

Returns:  numeric vector, empty numeric vector before requests made

Method status(): List HTTP status objects
Usage:
AsyncVaried$status()

Returns: a list of http_code objects, empty list before requests made

Method content(): Get raw content for each response

Usage:
AsyncVaried$content()

Returns: raw list, empty list before requests made

Method times(): curl request times

Usage:
AsyncVaried$times()

Returns: list of named numeric vectors, empty list before requests made

Method clone(): The objects of this class are cloneable with this method.

Usage:
AsyncVaried$clone(deep = FALSE)

Arguments:
deeplWhether to make a deep clone.

See Also
Other async: AsyncQueue, Async, HttpRequest

Examples

## Not run:
# pass in requests via ...
req1 <- HttpRequest$new(
  url = "https://httpbin.org/get",
  opts = list(verbose = TRUE),
  headers = list(foo = "bar")
)$get()
req2 <- HttpRequest$new(url = "https://httpbin.org/post")$post()

# Create an AsyncVaried object
out <- AsyncVaried$new(req1, req2)

# before you make requests, the methods return empty objects
out$status()
out$status_code()
out$content()
out$times()
out$parse()
out$responses()

# make requests
out$request()
# access various parts
## http status objects
out$status()
## status codes
out$status_code()
## content (raw data)
out$content()
## times
out$times()
## parsed content
out$parse()
## response objects
out$responses()

# use $verb() method to select http verb
method <- "post"
req1 <- HttpRequest$new(
  url = "https://httpbin.org/post",
  opts = list(verbose = TRUE),
  headers = list(foo = "bar")
)$verb(method)
req2 <- HttpRequest$new(url = "https://httpbin.org/post")$verb(method)
out <- AsyncVaried$new(req1, req2)
out
out$request()
out$responses()

# pass in requests in a list via .list param
reqlist <- list(
  HttpRequest$new(url = "https://httpbin.org/get")$get(),
  HttpRequest$new(url = "https://httpbin.org/post")$post(),
  HttpRequest$new(url = "https://httpbin.org/put")$put(),
  HttpRequest$new(url = "https://httpbin.org/delete")$delete(),
  HttpRequest$new(url = "https://httpbin.org/get?g=5")$get(),
  HttpRequest$new(
    url = "https://httpbin.org/post")$post(body = list(y = 9)),
  HttpRequest$new(
    url = "https://httpbin.org/get")$get(query = list(hello = "world"))
)
out <- AsyncVaried$new(.list = reqlist)
out$request()
out$status()
out$status_code()
out$content()
out$times()
out$parse()

# using auth with async
url <- "https://httpbin.org/basic-auth/user/passwd"
auth <- auth(user = "user", pwd = "passwd")
reqlist <- list(

"post"
req1 <- HttpRequest$new(
  url = "https://httpbin.org/post",
  opts = list(verbose = TRUE),
  headers = list(foo = "bar")
)$verb(method)
req2 <- HttpRequest$new(url = "https://httpbin.org/post")$verb(method)
out <- AsyncVaried$new(req1, req2)
out
out$request()
out$responses()

# pass in requests in a list via .list param
reqlist <- list(
  HttpRequest$new(url = "https://httpbin.org/get")$get(),
  HttpRequest$new(url = "https://httpbin.org/post")$post(),
  HttpRequest$new(url = "https://httpbin.org/put")$put(),
  HttpRequest$new(url = "https://httpbin.org/delete")$delete(),
  HttpRequest$new(url = "https://httpbin.org/get?g=5")$get(),
  HttpRequest$new(
    url = "https://httpbin.org/post")$post(body = list(y = 9)),
  HttpRequest$new(
    url = "https://httpbin.org/get")$get(query = list(hello = "world"))
)
out <- AsyncVaried$new(.list = reqlist)
out$request()
out$status()
out$status_code()
out$content()
out$times()
out$parse()

# using auth with async
url <- "https://httpbin.org/basic-auth/user/passwd"
auth <- auth(user = "user", pwd = "passwd")
reqlist <- list(

"post"
req1 <- HttpRequest$new(
  url = "https://httpbin.org/post",
  opts = list(verbose = TRUE),
  headers = list(foo = "bar")
)$verb(method)
req2 <- HttpRequest$new(url = "https://httpbin.org/post")$verb(method)
out <- AsyncVaried$new(req1, req2)
out
out$request()
out$responses()

# pass in requests in a list via .list param
reqlist <- list(
  HttpRequest$new(url = "https://httpbin.org/get")$get(),
  HttpRequest$new(url = "https://httpbin.org/post")$post(),
  HttpRequest$new(url = "https://httpbin.org/put")$put(),
  HttpRequest$new(url = "https://httpbin.org/delete")$delete(),
  HttpRequest$new(url = "https://httpbin.org/get?g=5")$get(),
  HttpRequest$new(
    url = "https://httpbin.org/post")$post(body = list(y = 9)),
  HttpRequest$new(
    url = "https://httpbin.org/get")$get(query = list(hello = "world"))
)
out <- AsyncVaried$new(.list = reqlist)
out$request()
out$status()
out$status_code()
out$content()
out$times()
out$parse()

# using auth with async
url <- "https://httpbin.org/basic-auth/user/passwd"
auth <- auth(user = "user", pwd = "passwd")
reqlist <- list(

"post"
req1 <- HttpRequest$new(
  url = "https://httpbin.org/post",
  opts = list(verbose = TRUE),
  headers = list(foo = "bar")
)$verb(method)
req2 <- HttpRequest$new(url = "https://httpbin.org/post")$verb(method)
out <- AsyncVaried$new(req1, req2)
out
out$request()
out$responses()
auth

Description

Authentication

Usage

auth(user, pwd, auth = "basic")

Arguments

user (character) username, required. see Details.
pwd (character) password, required. see Details.
auth (character) authentication type, one of basic (default), digest, digest_ie, gssnegotiate, ntlm, or any. required

Details

Only supporting simple auth for now, OAuth later maybe.

For user and pwd you are required to pass in some value. The value can be NULL to - which is equivalent to passing in an empty string like "" in http::authenticate. You may want to pass in NULL for both user and pwd for example if you are using gssnegotiate auth type. See example below.

Examples

auth(user = "foo", pwd = "bar", auth = "basic")
auth(user = "foo", pwd = "bar", auth = "digest")
auth(user = "foo", pwd = "bar", auth = "ntlm")
auth(user = "foo", pwd = "bar", auth = "any")

# gssnegotiate auth
auth(NULL, NULL, "gssnegotiate")

## Not run:
# with HttpClient
(res <- HttpClient$new(
  url = "https://httpbin.org/basic-auth/user/passwd",
  auth = auth(user = "user", pwd = "passwd")
))
res$auth
x <- res$get()
jsonlite::fromJSON(x$parse("UTF-8"))

# with HttpRequest
(res <- HttpRequest$new(
  url = "https://httpbin.org/basic-auth/user/passwd",
  auth = auth(user = "user", pwd = "passwd")
))
res$auth

## End(Not run)

Description

The HttpResponse class holds all the responses elements for an HTTP request. This document details how to work specifically with the content-type of the response headers
Content types

The "Content-Type" header in HTTP responses gives the media type of the response. The media type is both the data format and how the data is intended to be processed by a recipient. (modified from rfc7231)

Behavior of the parameters HttpResponse raise_for_ct* methods

- type: (only applicable for the raise_for_ct() method): instead of using one of the three other content type methods for html, json, or xml, you can specify a mime type to check, any of those in mime::mimemap
- charset: if you don't give a value to this parameter, we only check that the content type is what you expect; that is, the charset, if given, is ignored.
- behavior: by default when you call this method, and the content type does not match what the method expects, then we run stop() with a message. Instead of stopping, you can choose behavior="warning" and we'll throw a warning instead, allowing any downstream processing to proceed.

References

spec for content types: https://tools.ietf.org/html/rfc7231#section-3.1.1.5
spec for media types: https://tools.ietf.org/html/rfc7231#section-3.1.1.1

See Also

HttpResponse

Examples

```r
## Not run:
(x <- HttpClient$new(url = "https://httpbin.org"))
(res <- x$get())

## see the content type
res$response_headers

## check that the content type is text/html
res$raise_for_ct_html()

## it's def. not json
# res$raise_for_ct_json()

## give custom content type
res$raise_for_ct("text/html")
# res$raise_for_ct("application/json")
# res$raise_for_ct("foo/bar")

## check charset in addition to the media type
res$raise_for_ct_html(charset = "utf-8")
# res$raise_for_ct_html(charset = "utf-16")
```
cookies

# warn instead of stop
res$raise_for_ct_json(behavior = "warning")

## End(Not run)

### Working with cookies

**Description**

Working with cookies

**Examples**

```r
## Not run:
x <- HttpClient$new(
  url = "https://httpbin.org",
  opts = list(
    cookie = "c=1;f=5",
    verbose = TRUE
  )
)
x
# set cookies
(res <- x$get("cookies"))
jsonlite::fromJSON(res$parse("UTF-8"))

(x <- HttpClient$new(url = "https://httpbin.org"))
res <- x$get("cookies/set", query = list(foo = 123, bar = "ftw"))
jsonlite::fromJSON(res$parse("UTF-8"))
curl::handle_cookies(handle = res$handle)

# reuse handle
res2 <- x$get("get", query = list(hello = "world"))
jsonlite::fromJSON(res2$parse("UTF-8"))
curl::handle_cookies(handle = res2$handle)

# DOAJ
x <- HttpClient$new(url = "https://doaj.org")
res <- x$get("api/v1/journals/f3f2e7f23d444370ae5f5199f85bc100",
  verbose = TRUE)
res$response_headers`set-cookie`
curl::handle_cookies(handle = res$handle)
res2 <- x$get("api/v1/journals/9abfb36b06404e8a8566e1a44180bbdc",
  verbose = TRUE)

## reset handle
x$handle_pop()
## cookies no longer sent, as handle reset
```
res2 <- x$get("api/v1/journals/9abfb36b06404e8a8566e1a44180bbdc", verbose = TRUE)
## End(Not run)

---

**crul-options**  
*Set curl options, proxy, and basic auth*

**Description**  
Set curl options, proxy, and basic auth

**Usage**  
set_opts(...)  
setVerbose()  
set_proxy(x)  
set_auth(x)  
set_headers(...)  
crul_settings(reset = FALSE)

**Arguments**  
...  
For set_opts() any curl option in the set curl::curl_options(). For set_headers() a named list of headers  
x  
For set_proxy() a proxy object made with proxy(). For set_auth() an auth object made with auth()  
reset  
(logical) reset all settings (aka, delete them). Default: FALSE

**Details**  
- set_opts(): set curl options; supports any options in curl::curl_options()  
- setVerbose(): set custom curl verbose; sets verbose=TRUE and debugfunction to the callback result from curl::curl_options()  
- set_proxy(): set proxy settings, accepts proxy()  
- set_auth(): set authorization, accepts auth()  
- set_headers(): set request headers, a named list  
- crul_settings(): list all settings set via these functions

**Note**  
the mock option will be seen in output of crul_settings() but is set via the function mock()
Examples

```r
if (interactive()) {
  # get settings
crul_settings()

  # curl options
  set_opts(timeout_ms = 1000)
crul_settings()
  set_opts(timeout_ms = 4000)
crul_settings()
  set_opts(verb = TRUE)
crul_settings()
## Not run:
  HttpClient$new('https://httpbin.org')$get('get')
## End(Not run)

  # set_verbose - sets: `verbose=TRUE`, and `debugfunction` to
  # result of call to `curl_verbose()`, see `?curl_verbose`
  set_verbose()
crul_settings()

  # basic authentication
  set_auth(auth(user = "foo", pwd = "bar", auth = "basic"))
crul_settings()

  # proxies
  set_proxy(proxy("http://97.77.104.22:3128"))
crul_settings()

  # headers
  crul_settings(TRUE) # reset first
  set_headers(foo = "bar")
crul_settings()
  set_headers(User-Agent = "hello world")
crul_settings()
## Not run:
  set_opts(verb = TRUE)
  HttpClient$new('https://httpbin.org')$get('get')
## End(Not run)

  # reset
  crul_settings(TRUE)
crul_settings()

  # works with async functions
  # Async
  set_opts(verb = TRUE)
  cc <- Async$new(urls = c(
    'https://httpbin.org/get?a=5',
    'https://httpbin.org/get?foo=bar'))
  (res <- cc$get())
```
## AsyncVaried

```r
set_opts(verbos = TRUE)
set_headers(stuff = "things")
reqlist <- list(
  HttpRequest$new(url = "https://httpbin.org/get")$get(),
  HttpRequest$new(url = "https://httpbin.org/post")$post())
out <- AsyncVaried$new(.list = reqlist)
out$request()
```

### curl-options
curl options

---

**Description**

With the `opts` parameter you can pass in various curl options, including user agent string, whether to get verbose curl output or not, setting a timeout for requests, and more. See `curl::curl_options()` for all the options you can use. Note that you need to give curl options exactly as given in `curl::curl_options()`.

**Examples**

```r
## Not run:
url <- "https://httpbin.org"

# set curl options on client initialization
(res <- HttpClient$new(url = url, opts = list(verbos = TRUE)))
res$opts
res$get('get')

# or set curl options when performing HTTP operation
(res <- HttpClient$new(url = url))
res$get('get', verbos = TRUE)
res$get('get', stuff = "things")

# set a timeout
(res <- HttpClient$new(url = url, opts = list(timeout_ms = 1)))
# res$get('get')

# set user agent either as a header or an option
HttpClient$new(url = url,
  headers = list("User-Agent" = "hello world"),
  opts = list(verbos = TRUE))
}$get('get')

HttpClient$new(url = url,
  opts = list(verbos = TRUE, useragent = "hello world"))$_get('get')
```
# You can also set custom debug function via the verbose
# parameter when calling `HttpClient$new`
res <- HttpClient$new(url, verbose=curl_verbose())
res
res$gets("get")
res <- HttpClient$new(url, verbose=curl_verbose(data_in=TRUE))
res$gets("get")
res <- HttpClient$new(url, verbose=curl_verbose(info=TRUE))
res$gets("get")

## End(Not run)

<table>
<thead>
<tr>
<th>curl_verbose</th>
<th>curl verbose method</th>
</tr>
</thead>
</table>

**Description**

curl verbose method

**Usage**
curl_verbose(data_out = TRUE, data_in = FALSE, info = FALSE, ssl = FALSE)

**Arguments**

data_out Show data sent to the server
data_in Show data received from the server
info Show informational text from curl. This is mainly useful for debugging https and auth problems, so is disabled by default
ssl Show even data sent/received over SSL connections?

**Details**

line prefixes:

- * informative curl messages
- => headers sent (out)
- > data sent (out)
- *=> ssl data sent (out)
- <= headers received (in)
- < data received (in)
- <= ssl data received (in)

**Note**
adapted from htttr::verbose
#### handle

**Make a handle**

#### Description

Make a handle

#### Usage

```
handle(url, ...)
```

#### Arguments

- `url` (character) A url. required.
- `...` options passed on to `curl::new_handle()`

#### Examples

```
handle("https://httpbin.org")
```

```r
# handles - pass in your own handle
<< Not run:
  h <- handle("https://httpbin.org")
  (res <- HttpClient$new(handle = h))
  out <- res$get("get")

# End(Not run)
```

---

#### hooks

**Event Hooks**

#### Description

Trigger functions to run on requests and/or responses. See Details for more.

#### Details

Functions passed to request are run **before** the request occurs. The meaning of triggering a function on the request is that you can do things to the request object.

Functions passed to response are run **once** the request is done, and the response object is created. The meaning of triggering a function on the response is to do things on the response object.

The above for request and response applies the same whether you make real HTTP requests or mock with webmockr.

#### Note

Only supported on `HttpClient` for now
Examples

```r
## Not run:
# hooks on the request
fun_req <- function(request) {
  cat(paste0("Requesting: ", request$url$url), sep = "\n")
}
(x <- HttpClient$new(url = "https://httpbin.org",
  hooks = list(request = fun_req)))

x$hooks
x$hooks$request
r1 <- x$get('get')

captured_req <- list()
fun_req2 <- function(request) {
  cat("Capturing Request", sep = "\n")
captured_req <<- request
}
(x <- HttpClient$new(url = "https://httpbin.org",
  hooks = list(request = fun_req2)))

x$hooks
x$hooks$request
r1 <- x$get('get')
captured_req

# hooks on the response
fun_resp <- function(response) {
  cat(paste0("status_code: ", response$status_code), sep = "\n")
}
(x <- HttpClient$new(url = "https://httpbin.org",
  hooks = list(response = fun_resp)))

x$url
x$hooks
r1 <- x$get('get')

# both
(x <- HttpClient$new(url = "https://httpbin.org",
  hooks = list(request = fun_req, response = fun_resp)))

x$get("get")

## End(Not run)
```

---

**http-headers**

Working with HTTP headers

**Description**

Working with HTTP headers
## Examples

```r
## Not run:
(x <- HttpClient$new(url = "https://httpbin.org"))

# set headers
(res <- HttpClient$new(
  url = "https://httpbin.org",
  opts = list(
    verbose = TRUE
  ),
  headers = list(
    a = "stuff",
    b = "things"
  )
))
res$headers
# reassign header value
res$headers$a <- "that"
# define new header
res$headers$c <- "what"
# request
res$get('get')

## setting content-type via headers
(res <- HttpClient$new(
  url = "https://httpbin.org",
  opts = list(
    verbose = TRUE
  ),
  headers = list('Content-Type' = "application/json")
))
res$get('get')

## End(Not run)
```

---

**HttpClient**

**HTTP client**

### Description

Create and execute HTTP requests

### Value

- an [HttpResponse](#) object
handles

curl handles are re-used on the level of the connection object, that is, each HttpClient object is separate from one another so as to better separate connections.

If you don’t pass in a curl handle to the handle parameter, it gets created when a HTTP verb is called. Thus, if you try to get handle after creating a HttpClient object only passing url parameter, handle will be NULL. If you pass a curl handle to the handle parameter, then you can get the handle from the HttpClient object. The response from a http verb request does have the handle in the handle slot.

Public fields

url (character) a url
opts (list) named list of curl options
proxies a proxy() object
auth an auth() object
headers (list) named list of headers, see http-headers
handle a handle()
progress only supports httr::progress(), see progress
hooks a named list, see hooks

Methods

Public methods:

• HttpClient$print()
• HttpClient$new()
• HttpClient$get()
• HttpClient$post()
• HttpClient$put()
• HttpClient$patch()
• HttpClient$delete()
• HttpClient$head()
• HttpClient$verb()
• HttpClient$retry()
• HttpClient$handle_pop()
• HttpClient$url_fetch()
• HttpClient$clone()

Method print(): print method for HttpClient objects

Usage:
HttpClient$print(x, ...)

Arguments:
x self
... ignored
**Method** `new()`: Create a new `HttpClient` object

*Usage:*

```r
HttpClient$new(
  url,
  opts,
  proxies,
  auth,
  headers,
  handle,
  progress,
  hooks,
  verbose
)
```

*Arguments:*

- `url` (character) A url. One of `url` or `handle` required.
- `opts` any curl options
- `proxies` a `proxy()` object
- `auth` an `auth()` object
- `headers` named list of headers, see `http-headers`
- `handle` a `handle()`
- `progress` only supports `httr::progress()`, see `progress`
- `hooks` a named list, see `hooks`
- `verbose` a special handler for verbose curl output, accepts a function only. default is `NULL`. if used, `verbose` and `debugfunction` curl options are ignored if passed to `opts` on `$new()` and ignored if ... passed to a http method call
- `urls` (character) one or more URLs

*Returns:* A new `HttpClient` object

**Method** `get()`: Make a GET request

*Usage:*

```r
HttpClient$get(path = NULL, query = list(), disk = NULL, stream = NULL, ...)```

*Arguments:*

- `path` URL path, appended to the base URL
- `query` query terms, as a named list. any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted
- `disk` a path to write to. if `NULL` (default), memory used. See `curl::curl_fetch_disk()` for help.
- `stream` an R function to determine how to stream data. if `NULL` (default), memory used. See `curl::curl_fetch_stream()` for help
- ... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `httppost`, `post`, `postfields`, `postfields-size`, and customrequest

**Method** `post()`: Make a POST request
Usage:

```r
HttpClient$post(
  path = NULL,
  query = list(),
  body = NULL,
  disk = NULL,
  stream = NULL,
  encode = "multipart",
  ...
)
```

Arguments:

- **path**: URL path, appended to the base URL
- **query**: query terms, as a named list. Any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted.
- **body**: body as an R list
- **disk**: a path to write to. If NULL (default), memory used. See `curl::curl_fetch_disk()` for help.
- **stream**: an R function to determine how to stream data. If NULL (default), memory used. See `curl::curl_fetch_stream()` for help.
- **encode**: one of `form`, `multipart`, `json`, or `raw`

... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `httppost`, `post`, `postfields`, `postfield-size`, and `customrequest`.

Method **put()**: Make a PUT request

Usage:

```r
HttpClient$put(
  path = NULL,
  query = list(),
  body = NULL,
  disk = NULL,
  stream = NULL,
  encode = "multipart",
  ...
)
```

Arguments:

- **path**: URL path, appended to the base URL
- **query**: query terms, as a named list. Any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted.
- **body**: body as an R list
- **disk**: a path to write to. If NULL (default), memory used. See `curl::curl_fetch_disk()` for help.
- **stream**: an R function to determine how to stream data. If NULL (default), memory used. See `curl::curl_fetch_stream()` for help.
For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `htppost`, `post`, `postfields`, `postfield-size`, and `customrequest`.

**Method** `patch()`: Make a PATCH request

*Usage:*

```r
HttpClient$patch(
  path = NULL,
  query = list(),
  body = NULL,
  disk = NULL,
  stream = NULL,
  encode = "multipart",
  ...
)
```

*Arguments:*

- **path** URL path, appended to the base URL
- **query** query terms, as a named list. any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted
- **body** body as an R list
- **disk** a path to write to. if NULL (default), memory used. See `curl::curl_fetch_disk()` for help.
- **stream** an R function to determine how to stream data. if NULL (default), memory used. See `curl::curl_fetch_stream()` for help
- **encode** one of form, multipart, json, or raw

For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `htppost`, `post`, `postfields`, `postfield-size`, and `customrequest`.

**Method** `delete()`: Make a DELETE request

*Usage:*

```r
HttpClient$delete(
  path = NULL,
  query = list(),
  body = NULL,
  disk = NULL,
  stream = NULL,
  encode = "multipart",
  ...
)
```

*Arguments:*

- **path** URL path, appended to the base URL
query terms, as a named list. Any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted.

body body as an R list.

disk a path to write to. If NULL (default), memory used. See `curl::curl_fetch_disk()` for help.

stream an R function to determine how to stream data. If NULL (default), memory used. See `curl::curl_fetch_stream()` for help.

... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `httppost`, `post`, `postfields`, `postfield-size`, and `customrequest`.

Method `head()`: Make a HEAD request

Usage:
```
HttpClient$head(path = NULL, query = list(), ...)
```

Arguments:
- `path` URL path, appended to the base URL.
- `query` query terms, as a named list. Any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted.

... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `httppost`, `post`, `postfields`, `postfield-size`, and `customrequest`.

Method `verb()`: Use an arbitrary HTTP verb supported on this class. Supported verbs: "get", "post", "put", "patch", "delete", "head". Also supports retry.

Usage:
```
HttpClient$verb(verb, ...)
```

Arguments:
- `verb` an HTTP verb supported on this class: "get", "post", "put", "patch", "delete", "head". Also supports retry.

... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `httppost`, `post`, `postfields`, `postfield-size`, and `customrequest`.

Examples:
```
\dontrun{
  (x <- HttpClient$new(url = "https://httpbin.org"))
x$verb('get')
x$verb('GET')
x$verb('GET', query = list(foo = "bar"))
x$verb('retry', 'GET', path = "status/400")
}
```

Method `retry()`: Retry a request.
Usage:

```r
HttpClient$retry(
  verb,
  ..., 
  pause_base = 1,
  pause_cap = 60, 
  pause_min = 1, 
  times = 3, 
  terminate_on = NULL, 
  retry_only_on = NULL, 
  onwait = NULL
)
```

Arguments:

- **verb** an HTTP verb supported on this class: "get", "post", "put", "patch", "delete", "head". Also supports retry.
- ... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: httpget, httppost, post, postfields, postfield-size, and customrequest
- **pause_base, pause_cap, pause_min** basis, maximum, and minimum for calculating wait time for retry. Wait time is calculated according to the exponential backoff with full jitter algorithm. Specifically, wait time is chosen randomly between `pause_min` and the lesser of `pause_base * 2` and `pause_cap`, with `pause_base` doubling on each subsequent retry attempt. Use `pause_cap = Inf` to not terminate retrying due to cap of wait time reached.
- **times** the maximum number of times to retry. Set to `Inf` to not stop retrying due to exhausting the number of attempts.
- **terminate_on, retry_only_on** a vector of HTTP status codes. For `terminate_on`, the status codes for which to terminate retrying, and for `retry_only_on`, the status codes for which to retry the request.
- **onwait** a callback function if the request will be retried and a wait time is being applied. The function will be passed two parameters, the response object from the failed request, and the wait time in seconds. Note that the time spent in the function effectively adds to the wait time, so it should be kept simple.

Details: Retries the request given by `verb` until successful (HTTP response status < 400), or a condition for giving up is met. Automatically recognizes Retry-After and X-RateLimit-Reset headers in the response for rate-limited remote APIs.

Examples:

```r
\dontrun{
  x <- HttpClient$new(url = "https://httpbin.org")

  # retry, by default at most 3 times
  (res_get <- x$retry("GET", path = "status/400"))

  # retry, but not for 404 NOT FOUND
  (res_get <- x$retry("GET", path = "status/404", terminate_on = c(404)))
```
# retry, but only for exceeding rate limit (note that e.g. Github uses 403)
(res_get <- x$retry("GET", path = "status/429", retry_only_on = c(403, 429)))

**Method** handle_pop(): reset your curl handle

*Usage:*

HttpClient$handle_pop()

**Method** url_fetch(): get the URL that would be sent (i.e., before executing the request) the only things that change the URL are path and query parameters; body and any curl options don’t change the URL

*Usage:*

HttpClient$url_fetch(path = NULL, query = list())

*Arguments:*

- `path` URL path, appended to the base URL
- `query` query terms, as a named list. any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted

*Returns:* URL (character)

*Examples:*

```r
x <- HttpClient$new(url = "https://httpbin.org")
x$url_fetch()
x$url_fetch('get')
x$url_fetch('post')
x$url_fetch('get', query = list(foo = "bar"))
```

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

HttpClient$clone(deep = FALSE)

*Arguments:*

- `deep` Whether to make a deep clone.

**Note**

A little quirk about **crl** is that because user agent string can be passed as either a header or a curl option (both lead to a User-Agent header being passed in the HTTP request), we return the user agent string in the request_headers list of the response even if you pass in a useragent string as a curl option. Note that whether you pass in as a header like User-Agent or as a curl option like useragent, it is returned as request_headers$User-Agent so at least accessing it in the request headers is consistent.

**See Also**

- `http-headers`, `writing-options`, `cookies`, `hooks`
Examples

```r
## Not run:
# set your own handle
(h <- handle("https://httpbin.org"))
(x <- HttpClient$new(handle = h))
  x$handle
  x$url
  (out <- x$get("get"))
  x$handle
  x$url
  class(out)
  out$handle
  out$request_headers
  out$response_headers
  out$response_headers_all

# if you just pass a url, we create a handle for you
# this is how most people will use HttpClient
(x <- HttpClient$new(url = "https://httpbin.org"))
  x$url
  x$handle # is empty, it gets created when a HTTP verb is called
  (r1 <- x$get('get'))
  x$url
  x$handle
  r1$url
  r1$handle
  r1$content
  r1$response_headers
  r1$parse()

  (res_get2 <- x$get('get', query = list(hello = "world")))
  res_get2$parse()
  library("jsonlite")
  jsonlite::fromJSON(res_get2$parse())

  # post request
  (res_post <- x$post('post', body = list(hello = "world")))

  # empty body request
  x$post('post')

  # put request
  (res_put <- x$put('put'))

  # delete request
  (res_delete <- x$delete('delete'))

  # patch request
  (res_patch <- x$patch('patch'))

  # head request
  (res_head <- x$head())
```
# query params are URL encoded for you, so DO NOT do it yourself
## if you url encode yourself, it gets double encoded, and that's bad
(x <- HttpClient$new(url = "https://httpbin.org"))
res <- x$get("get", query = list(a = 'hello world'))

# access intermediate headers in response_headers_all
x <- HttpClient$new("https://doi.org/10.1007/978-3-642-40455-9_52-1")
bb <- x$get()
bb$response_headers_all

## End(Not run)

## Method `HttpClient$verb`
(x <- HttpClient$new(url = "https://httpbin.org"))
x$verb('get')
x$verb('GET')
x$verb('GET', query = list(foo = "bar"))
x$verb('retry', 'GET', path = "status/400")

## End(Not run)

## Method `HttpClient$retry`
(x <- HttpClient$new(url = "https://httpbin.org"))
# retry, by default at most 3 times
(res_get <- x$retry("GET", path = "status/400"))

# retry, but not for 404 NOT FOUND
(res_get <- x$retry("GET", path = "status/404", terminate_on = c(404)))

# retry, but only for exceeding rate limit (note that e.g. Github uses 403)
(res_get <- x$retry("GET", path = "status/429", retry_only_on = c(403, 429)))

## End(Not run)

## Method `HttpClient$url_fetch`
(x <- HttpClient$new(url = "https://httpbin.org"))
x$url_fetch()
x$url_fetch('get')
x$url_fetch('post')
x$url_fetch('get', query = list(foo = "bar"))
HttpRequest

HTTP request object

Description

Create HTTP requests

Details

This R6 class doesn’t do actual HTTP requests as does HttpClient() - it is for building requests to use for async HTTP requests in AsyncVaried()

Note that you can access HTTP verbs after creating an HttpRequest object, just as you can with HttpClient. See examples for usage.

Also note that when you call HTTP verbs on a HttpRequest object you don’t need to assign the new object to a variable as the new details you’ve added are added to the object itself.

See HttpClient() for information on parameters.

Public fields

url (character) a url

opts (list) named list of curl options

proxies a proxy() object

auth an auth() object

headers (list) named list of headers, see http-headers

handle a handle()

progress only supports httr::progress(), see progress

payload resulting payload after request

Methods

Public methods:

- HttpRequest$print()
- HttpRequest$new()
- HttpRequest$get()
- HttpRequest$post()
- HttpRequest$put()
- HttpRequest$patch()
- HttpRequest$delete()
- HttpRequest$head()
- HttpRequest$verb()
- HttpRequest$method()
- HttpRequest$clone()
Method print(): print method for HttpRequest objects

Usage:
HttpRequest$print(x, ...)

Arguments:
x self
... ignored

Method new(): Create a new HttpRequest object

Usage:
HttpRequest$new(url, opts, proxies, auth, headers, handle, progress)

Arguments:
url (character) A url. One of url or handle required.
opts any curl options
proxies a proxy() object
auth an auth() object
headers named list of headers, see http-headers
handle a handle()
progress only supports http::progress(), see progress
urls (character) one or more URLs

Returns: A new HttpRequest object

Method get(): Define a GET request

Usage:
HttpRequest$get(path = NULL, query = list(), disk = NULL, stream = NULL, ...)

Arguments:
path URL path, appended to the base URL
query query terms, as a named list
disk a path to write to. if NULL (default), memory used. See curl::curl_fetch_disk() for help.
stream an R function to determine how to stream data. if NULL (default), memory used. See curl::curl_fetch_stream() for help
... curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method post(): Define a POST request

Usage:
HttpRequest$post(
  path = NULL,
  query = list(),
  body = NULL,
  disk = NULL,
  stream = NULL,
  encode = "multipart",
  ...
)
Arguments:
path  URL path, appended to the base URL
query query terms, as a named list
body  body as an R list
disk a path to write to. if NULL (default), memory used. See `curl::curl_fetch_disk()` for help.
stream an R function to determine how to stream data. if NULL (default), memory used. See `curl::curl_fetch_stream()` for help
encode one of form, multipart, json, or raw
... curl options, only those in the acceptable set from `curl::curl_options()` except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method put(): Define a PUT request

Usage:
HttpRequest$put(
  path = NULL,
  query = list(),
  body = NULL,
  disk = NULL,
  stream = NULL,
  encode = "multipart",
  ...
)

Arguments:
path  URL path, appended to the base URL
query query terms, as a named list
body  body as an R list
disk a path to write to. if NULL (default), memory used. See `curl::curl_fetch_disk()` for help.
stream an R function to determine how to stream data. if NULL (default), memory used. See `curl::curl_fetch_stream()` for help
encode one of form, multipart, json, or raw
... curl options, only those in the acceptable set from `curl::curl_options()` except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method patch(): Define a PATCH request

Usage:
HttpRequest$patch(
  path = NULL,
  query = list(),
  body = NULL,
  disk = NULL,
  stream = NULL,
  encode = "multipart",
  ...
)
HttpRequest

Arguments:
path  URL path, appended to the base URL
query query terms, as a named list
body  body as an R list
disk a path to write to. if NULL (default), memory used. See curl::curl_fetch_disk() for help.
stream an R function to determine how to stream data. if NULL (default), memory used. See curl::curl_fetch_stream() for help
encode one of form, multipart, json, or raw
... curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method delete(): Define a DELETE request
Usage:
HttpRequest$delete(
  path = NULL,
  query = list(),
  body = NULL,
  disk = NULL,
  stream = NULL,
  encode = "multipart",
  ...
)

Arguments:
path  URL path, appended to the base URL
query query terms, as a named list
body  body as an R list
disk a path to write to. if NULL (default), memory used. See curl::curl_fetch_disk() for help.
stream an R function to determine how to stream data. if NULL (default), memory used. See curl::curl_fetch_stream() for help
encode one of form, multipart, json, or raw
... curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method head(): Define a HEAD request
Usage:
HttpRequest$head(path = NULL, ...)

Arguments:
path  URL path, appended to the base URL
... curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Method verb(): Use an arbitrary HTTP verb supported on this class Supported verbs: get, post, put, patch, delete, head
HttpRequest

Usage:
HttpRequest$verb(verb, ...)

Arguments:
verb  an HTTP verb supported on this class: get, post, put, patch, delete, head. Also supports retry.
...  curl options, only those in the acceptable set from curl::curl_options() except the following: httpget, httppost, post, postfields, postfieldsize, and customrequest

Examples:
z <- HttpRequest$new(url = "https://httpbin.org/get")
res <- z$verb('get', query = list(hello = "world"))
res$payload

Method method(): Get the HTTP method (if defined)

Usage:
HttpRequest$method()

Returns: (character) the HTTP method

Method clone(): The objects of this class are cloneable with this method.

Usage:
HttpRequest$clone(deep = FALSE)

Arguments:
deep  Whether to make a deep clone.

See Also

http-headers, writing-options

Other async: AsyncQueue, AsyncVaried, Async

Examples

## Not run:
x <- HttpRequest$new(url = "https://httpbin.org/get")
## note here how the HTTP method is shown on the first line to the right
x$get()

## assign to a new object to keep the output
z <- x$get()
### get the HTTP method
z$method()

(x <- HttpRequest$new(url = "https://httpbin.org/get"))$get()
x$url
x$payload

(x <- HttpRequest$new(url = "https://httpbin.org/post"))
x$post(body = list(foo = "bar"))
HttpRequest$new(
  url = "https://httpbin.org/get",
  headers = list(
    'Content-Type' = "application/json"
  )
)

## End(Not run)

## ------------------------------------------------
## Method `HttpRequest$verb`
## ------------------------------------------------

z <- HttpRequest$new(url = "https://httpbin.org/get")
res <- z$verb('get', query = list(hello = "world"))
res$payload

---

## Base HTTP response object

### Description

Class with methods for handling HTTP responses

### Details

#### Additional Methods

- **raise_for_ct**(type, charset = NULL, behavior = "stop") Check response content-type; stop or warn if not matched. Parameters:
  - type: (character) a mime type to match against; see mime::mimemap for allowed values
  - charset: (character) if a charset string given, we check that it matches the charset in the content type header. default: NULL
  - behavior: (character) one of stop (default) or warning

- **raise_for_ct_html**(charset = NULL, behavior = "stop") Check that the response content-type is text/html; stop or warn if not matched. Parameters: see raise_for_ct()

- **raise_for_ct_json**(charset = NULL, behavior = "stop") Check that the response content-type is application/json; stop or warn if not matched. Parameters: see raise_for_ct()

- **raise_for_ct_xml**(charset = NULL, behavior = "stop") Check that the response content-type is application/xml; stop or warn if not matched. Parameters: see raise_for_ct()

### Public fields

- **method** (character) one or more URLs
- **url** (character) one or more URLs
- **opts** (character) one or more URLs
handle (character) one or more URLs
status_code (character) one or more URLs
request_headers (character) one or more URLs
response_headers (character) one or more URLs
response_headers_all (character) one or more URLs
modified (character) one or more URLs
times (character) one or more URLs
content (character) one or more URLs
request (character) one or more URLs
raise_for_ct for ct method (general)
raise_for_ct_html for ct method (html)
raise_for_ct_json for ct method (json)
raise_for_ct_xml for ct method (xml)

Methods

Public methods:

- HttpResponse$print()
- HttpResponse$new()
- HttpResponse$parse()
- HttpResponse$success()
- HttpResponse$status_http()
- HttpResponse$raise_for_status()
- HttpResponse$clone()

Method print(): print method for HttpResponse objects

Usage:
HttpResponse$print(x, ...)

Arguments:
x self
... ignored

Method new(): Create a new HttpResponse object

Usage:
HttpResponse$new(
    method,
    url,
    opts,
    handle,
    status_code,
    request_headers,
    response_headers,
response_headers_all,
modified,
times,
content,
request
)

Arguments:
method (character) HTTP method
url (character) A url, required
opts (list) curl options
handle A handle
status_code (integer) status code
request_headers (list) request headers, named list
response_headers (list) response headers, named list
response_headers_all (list) all response headers, including intermediate redirect headers,
unnamed list of named lists
modified (character) modified date
times (vector) named vector
content (raw) raw binary content response
request request object, with all details

Method parse(): Parse the raw response content to text

Usage:
HttpResponse$parse(encoding = NULL, ...)

Arguments:
encoding (character) A character string describing the current encoding. If left as NULL, we
attempt to guess the encoding. Passed to from parameter in iconv
... additional parameters passed on to iconv (options: sub, mark, toRaw). See ?iconv for
help

Returns: character string

Method success(): Was status code less than or equal to 201

Usage:
HttpResponse$success()

Returns: boolean

Method status_http(): Get HTTP status code, message, and explanation

Usage:
HttpResponse$status_http( verbose = FALSE)

Arguments:
verbose (logical) whether to get verbose http status description, default: FALSE

Returns: object of class "http_code", a list with slots for status_code, message, and explanation
**Method** `raise_for_status()`: Check HTTP status and stop with appropriate HTTP error code and message if \( \geq 300 \). otherwise use `httpcode`. If you have `fauxpas` installed we use that.

*Usage:*
```
HttpResponse$raise_for_status()
```

*Returns:* stop or warn with message

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*
```
HttpResponse$clone(deep = FALSE)
```

*Arguments:*
- `deep` Whether to make a deep clone.

**See Also**

`content-types`

**Examples**

```r
## Not run:
x <- HttpResponse$new(method = "get", url = "https://httpbin.org")
x$url
x$method

x <- HttpClient$new(url = "https://httpbin.org")
(res <- x$get())
res$request_headers
res$response_headers
res$parse()
res$status_code
res$status_http()
res$status_http()$status_code
res$status_http()$message
res$status_http()$explanation
res$succeess()

x <- HttpClient$new(url = "https://httpbin.org/status/404")
(res <- x$get())
# res$raise_for_status()

x <- HttpClient$new(url = "https://httpbin.org/status/414")
(res <- x$get())
# res$raise_for_status()

## End(Not run)
```
Mocking HTTP requests

### Usage

mock(on = TRUE)

#### Arguments

- **on** (logical) turn mocking on with TRUE or turn off with FALSE. By default is FALSE

### Details

webmockr package required for mocking behavior

### Examples

```r
## Not run:
if (interactive()) {
  # load webmockr
  library(webmockr)
  library(crul)

  URL <- "https://httpbin.org"

  # turn on mocking
  crul::mock()

  # stub a request
  stub_request("get", file.path(URL, "get"))
  webmockr::webmockr_stub_registry

  # create an HTTP client
  (x <- HttpClient$new(url = URL))

  # make a request - matches stub - no real request made
  x$get('get')

  # allow net connect
  webmockr::webmockr_allow_net_connect()
  x$get('get', query = list(foo = "bar"))
  webmockr::webmockr_disable_net_connect()
  x$get('get', query = list(foo = "bar"))
}
```
## End(Not run)

### ok

#### check if a url is okay

**Description**

check if a url is okay

**Usage**

```r
ok(x, status = 200L, info = TRUE, verb = "head", ua_random = FALSE, ...)
```

**Arguments**

- **x**: either a URL as a character string, or an object of class `HttpContext`
- **status**: (integer) one or more HTTP status codes, must be integers. default: `200L`, since this is the most common signal that a URL is okay, but there may be cases in which your URL is okay if it’s a `201L`, or some other status code.
- **info**: (logical) in the case of an error, do you want a `message()` about it? Default: `TRUE`
- **verb**: (character) use "head" (default) or "get" HTTP verb for the request. note that "get" will take longer as it returns a body. however, "verb=get" may be your only option if a url blocks head requests
- **ua_random**: (logical) use a random user agent string? default: `TRUE`. if you set `useragent` curl option it will override this setting. The random user agent string is pulled from a vector of 50 user agent strings generated from `charlatan::UserAgentProvider` (by executing `replicate(30,UserAgentProvider$new()$user_agent())`)
- **...**: args passed on to `HttpContext`

**Details**

We internally verify that status is an integer and in the known set of HTTP status codes, and that info is a boolean

You may have to fiddle with the parameters to `ok()` as well as curl options to get the "right answer". If you think you are getting incorrectly getting `FALSE`, the first thing to do is to pass in `verbose=TRUE` to `ok()`. That will give you verbose curl output and will help determine what the issue may be. Here's some different scenarios:

- the site blocks head requests: some sites do this, try `verb="get"
- it will be hard to determine a site that requires this, but it’s worth trying a random useragent string, e.g., `ok(useragent = "foobar")`
- some sites are up and reachable but you could get a 403 Unauthorized error, there’s nothing you can do in this case other than having access
• it's possible to get a weird HTTP status code, e.g., LinkedIn gives a 999 code, they're trying to prevent any programmatic access

A FALSE result may be incorrect depending on the use case. For example, if you want to know if curl based scraping will work without fiddling with curl options, then the FALSE is probably correct, but if you want to fiddle with curl options, then first step would be to send verbose=TRUE to see what's going on with any redirects and headers. You can set headers, user agent strings, etc. to get closer to the request you want to know about. Note that a user agent string is always passed by default, but it may not be the one you want.

Value

a single boolean, if TRUE the URL is up and okay, if FALSE it is down; but, see Details

Examples

```r
## Not run:
# 200
ok("https://www.google.com")
# 200
ok("https://httpbin.org/status/200")
# more than one status
ok("https://www.google.com", status = c(200L, 202L))
# 404
ok("https://httpbin.org/status/404")
# doesn't exist
ok("https://stuff.bar")
# doesn't exist
ok("stuff")

# use get verb instead of head
ok("http://animalnexus.ca")
ok("http://animalnexus.ca", verb = "get")

# some urls will require a different useragent string
# they probably regex the useragent string
ok("https://doi.org/10.1093/chemse/bjq042")
ok("https://doi.org/10.1093/chemse/bjq042", verb = "get", useragent = "foobar")

# with random user agent's
## here, use a request hook to print out just the user agent string so
## we can see what user agent string is being sent off
fun_ua <- function(request) {
  message(paste0("User-agent: ", request$options$useragent), sep = "\n")
}
z <- crul::HttpClient$new("https://doi.org/10.1093/chemse/bjq042",
  hooks = list(request = fun_ua))
z
```

# with HttpClient

```r
z <- crul::HttpClient$new("https://httpbin.org/status/404", 
  opts = list(verbose = TRUE))
ok(z)

## End(Not run)
```

---

## Paginator

### Paginator client

## Description

A client to help you paginate

## Details

See `HttpClient()` for information on parameters

## Value

a list, with objects of class `HttpResponse()`. Responses are returned in the order they are passed in.

## Methods to paginate

Supported now:

- **limit_offset**: the most common way (in my experience), so is the default. This method involves setting how many records and what record to start at for each request. We send these query parameters for you.
- **page_perpage**: set the page to fetch and (optionally) how many records to get per page

Supported later, hopefully:

- **link_headers**: link headers are URLs for the next/previous/last request given in the response header from the server. This is relatively uncommon, though is recommended by JSONAPI and is implemented by a well known API (GitHub).
- **cursor**: this works by a single string given back in each response, to be passed in the subsequent response, and so on until no more records remain. This is common in Solr

## Public fields

- **http_req**: an object of class `HttpClient` by (character) how to paginate. Only 'limit_offset' supported for now. In the future will support 'link_headers' and 'cursor'. See Details.
- **chunk**: (numeric/integer) the number by which to chunk requests, e.g., 10 would be be each request gets 10 records. number is passed through `format()` to prevent larger numbers from being scientifically formatted
limit_param (character) the name of the limit parameter. Default: limit
offset_param (character) the name of the offset parameter. Default: offset
limit (numeric/integer) the maximum records wanted. number is passed through format() to prevent larger numbers from being scientifically formatted
page_param (character) the name of the page parameter. Default: NULL
per_page_param (character) the name of the per page parameter. Default: NULL
progress (logical) print a progress bar, using utils::txtProgressBar. Default: FALSE.

Methods

Public methods:
• Paginator$print()
• Paginator$new()
• Paginator$get()
• Paginator$post()
• Paginator$put()
• Paginator$patch()
• Paginator$delete()
• Paginator$head()
• Paginator$responses()
• Paginator$status_code()
• Paginator$status()
• Paginator$parse()
• Paginator$content()
• Paginator$times()
• Paginator$url_fetch()
• Paginator$clone()

Method print(): print method for Paginator objects

Usage:
Paginator$print(x, ...)

Arguments:
x self
... ignored

Method new(): Create a new Paginator object

Usage:
Paginator$new(
  client,
  by = "limit_offset",
  limit_param = NULL,
  offset_param = NULL,
  limit = NULL,
chunk = NULL,
page_param = NULL,
per_page_param = NULL,
progress = FALSE
)

Arguments:
client an object of class HttpClient, from a call to HttpClient
by (character) how to paginate. Only 'limit_offset' supported for now. In the future will support
'link_headers' and 'cursor'. See Details.
limit_param (character) the name of the limit parameter. Default: limit
offset_param (character) the name of the offset parameter. Default: offset
limit (numeric/integer) the maximum records wanted
chunk (numeric/integer) the number by which to chunk requests, e.g., 10 would be each
request gets 10 records
page_param (character) the name of the page parameter.
per_page_param (character) the name of the per page parameter.
progress (logical) print a progress bar, using utils::txtProgressBar. Default: FALSE.

Returns: A new Paginator object

Method get(): make a paginated GET request

Usage:
Paginator$get(path = NULL, query = list(), ...)

Arguments:
path URL path, appended to the base URL
query query terms, as a named list. any numeric values are passed through format() to prevent
larger numbers from being scientifically formatted
... For retry, the options to be passed on to the method implementing the requested verb,
including curl options. Otherwise, curl options, only those in the acceptable set from
curl::curl_options() except the following: httpget, httppost, post, postfields, postfield-size, and customrequest

Method post(): make a paginated POST request

Usage:
Paginator$post(
  path = NULL,
  query = list(),
  body = NULL,
  encode = "multipart",
  ...
)

Arguments:
path URL path, appended to the base URL
query query terms, as a named list. any numeric values are passed through format() to prevent
larger numbers from being scientifically formatted
body  body as an R list
encode one of form, multipart, json, or raw
... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `httppost`, `post`, `postfields`, `postfield-size`, and `customrequest`

**Method** `put()`: make a paginated PUT request

*Usage:*

```r
Paginator$put(
  path = NULL,
  query = list(),
  body = NULL,
  encode = "multipart",
  ...
)
```

*Arguments:*

- `path` URL path, appended to the base URL
- `query` query terms, as a named list. any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted
- `body` body as an R list
- `encode` one of form, multipart, json, or raw

... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `httppost`, `post`, `postfields`, `postfield-size`, and `customrequest`

**Method** `patch()`: make a paginated PATCH request

*Usage:*

```r
Paginator$patch(
  path = NULL,
  query = list(),
  body = NULL,
  encode = "multipart",
  ...
)
```

*Arguments:*

- `path` URL path, appended to the base URL
- `query` query terms, as a named list. any numeric values are passed through `format()` to prevent larger numbers from being scientifically formatted
- `body` body as an R list
- `encode` one of form, multipart, json, or raw

... For retry, the options to be passed on to the method implementing the requested verb, including curl options. Otherwise, curl options, only those in the acceptable set from `curl::curl_options()` except the following: `httpget`, `httppost`, `post`, `postfields`, `postfield-size`, and `customrequest`
Method delete(): make a paginated DELETE request

Usage:
Paginator$delete(
  path = NULL,
  query = list(),
  body = NULL,
  encode = "multipart",
  ...
)

Arguments:
  path URL path, appended to the base URL
  query query terms, as a named list. any numeric values are passed through format() to prevent
        larger numbers from being scientifically formatted
  body body as an R list
  encode one of form, multipart, json, or raw

... For retry, the options to be passed on to the method implementing the requested verb,
including curl options. Otherwise, curl options, only those in the acceptable set from
  curl::curl_options() except the following: httpget, httppost, post, postfields, postfield-
  size, and customrequest

Method head(): make a paginated HEAD request

Usage:
Paginator$head(path = NULL, ...)

Arguments:
  path URL path, appended to the base URL

... For retry, the options to be passed on to the method implementing the requested verb,
including curl options. Otherwise, curl options, only those in the acceptable set from
  curl::curl_options() except the following: httpget, httppost, post, postfields, postfield-
  size, and customrequest

Details: not sure if this makes any sense or not yet

Method responses(): list responses

Usage:
Paginator$responses()

Returns: a list of HttpResponse objects, empty list before requests made

Method status_code(): Get HTTP status codes for each response

Usage:
Paginator$status_code()

Returns: numeric vector, empty numeric vector before requests made

Method status(): List HTTP status objects

Usage:
Paginator$status()
Returns: a list of http_code objects, empty list before requests made

Method parse(): parse content
Usage:
Paginator$parse(encoding = "UTF-8")
Arguments:
encoding (character) the encoding to use in parsing. default:"UTF-8"
Returns: character vector, empty character vector before requests made

Method content(): Get raw content for each response
Usage:
Paginator$content()
Returns: raw list, empty list before requests made

Method times(): curl request times
Usage:
Paginator$times()
Returns: list of named numeric vectors, empty list before requests made

Method url_fetch(): get the URL that would be sent (i.e., before executing the request) the only things that change the URL are path and query parameters; body and any curl options don’t change the URL
Usage:
Paginator$url_fetch(path = NULL, query = list())
Arguments:
path URL path, appended to the base URL
query query terms, as a named list. any numeric values are passed through format() to prevent larger numbers from being scientifically formatted
Returns: URLs (character)
Examples:
\dontrun{
  cli <- HttpClient$new(url = "https://api.crossref.org")
  cc <- Paginator$new(client = cli, limit_param = "rows",
            offset_param = "offset", limit = 50, chunk = 10)
  cc$url_fetch('works')
  cc$url_fetch('works', query = list(query = "NSF"))
}

Method clone(): The objects of this class are cloneable with this method.
Usage:
Paginator$clone(deep = FALSE)
Arguments:
deep Whether to make a deep clone.
Examples

## Not run:
if (interactive()) {
# limit/offset approach
con <- HttpClient$new(url = "https://api.crossref.org")
cc <- Paginator$new(client = con, limit_param = "rows",
                   offset_param = "offset", limit = 50, chunk = 10)
cc
c$c$get('works')
cc
c$c$responses()
cc$status()
cc$status_code()
cc$times()
# cc$content()
cc$parse()
lapply(cc$parse(), jsonlite::fromJSON)
}

# page/per page approach (with no per_page param allowed)
conn <- HttpClient$new(url = "https://discuss.ropensci.org")
cc <- Paginator$new(client = conn, by = "page_perpage",
                   page_param = "page", per_page_param = "per_page", limit = 90, chunk = 30)
cc
c$c$get('c/usecases/1/latest.json')
c$c$responses()
lapply(cc$parse(), jsonlite::fromJSON)

# page/per_page
conn <- HttpClient$new('https://api.inaturalist.org')
cc <- Paginator$new(conn, by = "page_perpage", page_param = "page",
                   per_page_param = "per_page", limit = 90, chunk = 30)
cc
c$c$get('v1/observations', query = list(taxon_name="Helianthus"))
c$c$responses()
res <- lapply(cc$parse(), jsonlite::fromJSON)
res[[1]]$total_results
vapply(res, "{[", 1L, "page")
vapply(res, "{[", 1L, "per_page")
vapply(res, function(w) NROW(w$results), 1L)
## another
ccc <- Paginator$new(conn, by = "page_perpage", page_param = "page",
                   per_page_param = "per_page", limit = 500, chunk = 30, progress = TRUE)
ccc
c$cc$get('v1/observations', query = list(taxon_name="Helianthus"))
res2 <- lapply(ccc$parse(), jsonlite::fromJSON)
vapply(res2, function(w) NROW(w$results), 1L)

# progress bar
(con <- HttpClient$new(url = "https://api.crossref.org"))
cc <- Paginator$new(client = con, limit_param = "rows",
                   offset_param = "offset", limit = 50, chunk = 10,
progress = TRUE)
## Method `Paginator$url_fetch`

```r
ccli <- HttpClient$new(url = "https://api.crossref.org")
cc <- Paginator$new(client = cli, limit_param = "rows", offset_param = "offset", limit = 50, chunk = 10)
cc$url_fetch('works')
cc$url_fetch('works', query = list(query = "NSF"))
```

## Not run

### Description

progress bars

### Details

pass `httr::progress()` to progress param in `HttpClient`, which pulls out relevant info to pass down to `curl`

- if file sizes known you get progress bar; if file sizes not known you get bytes downloaded

See the README for examples

### Proxy Options

#### Description

proxy options

#### Usage

```r
proxy(url, user = NULL, pwd = NULL, auth = "basic")
```
Arguments

url  (character) URL, with scheme (http/https), domain and port (must be numeric). required.
user  (character) username, optional
pwd  (character) password, optional
auth  (character) authentication type, one of basic (default), digest, digest_ie, gssnegotiate, ntlm, any or NULL. optional

Details

See https://www.hidemyass.com/proxy for a list of proxies you can use

Examples

proxy("http://97.77.104.22:3128")
proxy("97.77.104.22:3128")
proxy("http://97.77.104.22:3128", "foo", "bar")
proxy("http://97.77.104.22:3128", "foo", "bar", auth = "digest")
proxy("http://97.77.104.22:3128", "foo", "bar", auth = "ntlm")

# socks
proxy("socks5://localhost:9050/", auth = NULL)

## Not run:
# with proxy (look at request/outgoing headers)
# (res <- HttpClient$new(
#  url = "http://www.google.com",
#  proxies = proxy("http://97.77.104.22:3128")
# ))
# res$proxies
# res$get(verbse = TRUE)

# vs. without proxy (look at request/outgoing headers)
# (res2 <- HttpClient$new(url = "http://www.google.com"))
# res2$GET(verbse = TRUE)

# Use authentication
# (res <- HttpClient$new(
#  url = "http://google.com",
#  proxies = proxy("http://97.77.104.22:3128", user = "foo", pwd = "bar")
# ))

# another example
# (res <- HttpClient$new(
#  url = "http://ip.tyk.nu/",
#  proxies = proxy("http://200.29.191.149:3128")
# ))
# res$get()$parse("UTF-8")

## End(Not run)
Description
upload file

Usage
upload(path, type = NULL)

Arguments
path  (character) a single path, file must exist
type  (character) a file type, guessed by mime::guess_type if not given

Examples
## Not run:
# image
path <- file.path(Sys.getenv("R_DOC_DIR"), "html/logo.jpg")
(x <- HttpClient$new(url = "https://eu.httpbin.org"))
res <- x$post(path = "post", body = list(y = upload(path)))
res$content

# text file, in a list
file <- upload(system.file("CITATION"))
res <- x$post(path = "post", body = list(y = file))
jsonlite::fromJSON(res$parse("UTF-8"))

# text file, as data
res <- x$post(path = "post", body = file)
jsonlite::fromJSON(res$parse("UTF-8"))

## End(Not run)

url_build  Build and parse URLs

Description
Build and parse URLs

Usage
url_build(url, path = NULL, query = NULL)

url_parse(url)
Arguments

url (character) a url, length 1
path (character) a path, length 1
query (list) a named list of query parameters

Value

url_build returns a character string URL; url_parse returns a list with URL components

Examples

url_build("https://httpbin.org")
url_build("https://httpbin.org", "get")
url_build("https://httpbin.org", "post")
url_build("https://httpbin.org", "get", list(foo = "bar"))

url_parse("httpbin.org")
url_parse("http://httpbin.org")
url_parse(url = "https://httpbin.org")
url_parse("https://httpbin.org/get")
url_parse("https://httpbin.org/get?foo=bar")
url_parse("https://httpbin.org/get?foo=bar&stuff=things")
url_parse("https://httpbin.org/get?foo=bar&stuff=things[]")

verb-DELETE

HTTP verb info: DELETE

Description

The DELETE method deletes the specified resource.

The DELETE method

The DELETE method requests that the origin server remove the association between the target resource and its current functionality. In effect, this method is similar to the rm command in UNIX: it expresses a deletion operation on the URI mapping of the origin server rather than an expectation that the previously associated information be deleted.

See https://tools.ietf.org/html/rfc7231#section-4.3.5 for further details.

References

https://tools.ietf.org/html/rfc7231#section-4.3.5

See Also

crul-package

Other verbs: verb-GET, verb-HEAD, verb-PATCH, verb-POST, verb-PUT
Examples

```r
## Not run:
x <- HttpClient$new(url = "https://httpbin.org")
x$delete(path = 'delete')

## a list
(res1 <- x$delete('delete', body = list(hello = "world"), verbose = TRUE))
jsonlite::fromJSON(res1$parse("UTF-8"))

## a string
(res2 <- x$delete('delete', body = "hello world", verbose = TRUE))
jsonlite::fromJSON(res2$parse("UTF-8"))

## empty body request
x$delete('delete', verbose = TRUE)

## End(Not run)
```

Description

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

The GET method

The GET method requests transfer of a current selected representation for the target resource. GET is the primary mechanism of information retrieval and the focus of almost all performance optimizations. Hence, when people speak of retrieving some identifiable information via HTTP, they are generally referring to making a GET request.

It is tempting to think of resource identifiers as remote file system pathnames and of representations as being a copy of the contents of such files. In fact, that is how many resources are implemented (see Section 9.1 ([https://tools.ietf.org/html/rfc7231#section-9.1](https://tools.ietf.org/html/rfc7231#section-9.1)) for related security considerations). However, there are no such limitations in practice. The HTTP interface for a resource is just as likely to be implemented as a tree of content objects, a programmatic view on various database records, or a gateway to other information systems. Even when the URI mapping mechanism is tied to a file system, an origin server might be configured to execute the files with the request as input and send the output as the representation rather than transfer the files directly. Regardless, only the origin server needs to know how each of its resource identifiers corresponds to an implementation and how each implementation manages to select and send a current representation of the target resource in a response to GET.

A client can alter the semantics of GET to be a "range request", requesting transfer of only some part(s) of the selected representation, by sending a Range header field in the request (RFC7233: [https://tools.ietf.org/html/rfc7233](https://tools.ietf.org/html/rfc7233)).
A payload within a GET request message has no defined semantics; sending a payload body on a GET request might cause some existing implementations to reject the request.

The response to a GET request is cacheable; a cache MAY use it to satisfy subsequent GET and HEAD requests unless otherwise indicated by the Cache-Control header field (Section 5.2 of RFC7234: https://tools.ietf.org/html/rfc7234#section-5.2).

References

https://tools.ietf.org/html/rfc7231#section-4.3.1

See Also

crul-package

Other verbs: verb-DELETE, verb-HEAD, verb-PATCH, verb-POST, verb-PUT

Examples

```r
## Not run:
x <- HttpClient$new(url = "https://httpbin.org")
x$get(path = 'get')
## End(Not run)
```

---

**verb-HEAD**

**HTTP verb info: HEAD**

Description

The HEAD method asks for a response identical to that of a GET request, but without the response body.

The HEAD method

The HEAD method is identical to GET except that the server MUST NOT send a message body in the response (i.e., the response terminates at the end of the header section). The server SHOULD send the same header fields in response to a HEAD request as it would have sent if the request had been a GET, except that the payload header fields MAY be omitted. This method can be used for obtaining metadata about the selected representation without transferring the representation data and is often used for testing hypertext links for validity, accessibility, and recent modification.

See https://tools.ietf.org/html/rfc7231#section-4.3.2 for further details.

References

https://tools.ietf.org/html/rfc7231#section-4.3.2
verb-PATCH

See Also

crut-package

Other verbs: verb-DELETE, verb-GET, verb-PATCH, verb-POST, verb-PUT

Examples

```r
## Not run:
x <- HttpClient$new(url = "https://httpbin.org")
x$head()

## End(Not run)
```

---

verb-PATCH  HTTP verb info: PATCH

Description

The PATCH method is used to apply partial modifications to a resource.

The PATCH method

The PATCH method requests that a set of changes described in the request entity be applied to the resource identified by the Request-URI. The set of changes is represented in a format called a "patch document" identified by a media type. If the Request-URI does not point to an existing resource, the server MAY create a new resource, depending on the patch document type (whether it can logically modify a null resource) and permissions, etc.


References


See Also

crut-package

Other verbs: verb-DELETE, verb-GET, verb-HEAD, verb-POST, verb-PUT

Examples

```r
## Not run:
x <- HttpClient$new(url = "https://httpbin.org")
x$patch(path = 'patch', body = list(hello = "mars"))

## End(Not run)
```
**HTTP verb info: POST**

**Description**

The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

**The POST method**

If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server SHOULD send a 201 (Created) response containing a Location header field that provides an identifier for the primary resource created (Section 7.1.2 [https://tools.ietf.org/html/rfc7231#section-7.1.2](https://tools.ietf.org/html/rfc7231#section-7.1.2)) and a representation that describes the status of the request while referring to the new resource(s).

See [https://tools.ietf.org/html/rfc7231#section-4.3.3](https://tools.ietf.org/html/rfc7231#section-4.3.3) for further details.

**References**

[https://tools.ietf.org/html/rfc7231#section-4.3.3](https://tools.ietf.org/html/rfc7231#section-4.3.3)

**See Also**

`crul-package`

**Other verbs:** `verb-DELETE`, `verb-GET`, `verb-HEAD`, `verb-PATCH`, `verb-PUT`

**Examples**

```r
## Not run:
x <- HttpClient$new(url = "https://httpbin.org")

# a named list
x$post(path='post', body = list(hello = "world"))

# a string
x$post(path='post', body = "hello world")

# an empty body request
x$post(path='post')

# encode="form"
res <- x$post(path="post",
   encode = "form",
   body = list(
      custname = 'Jane',
      custtel = '444-4444',
      size = 'small',
      topping = 'bacon',
   ))
```
verb-PUT

```r

comments = 'make it snappy'

jsonlite::fromJSON(res$parse("UTF-8"))

# encode="json"
res <- x$post("post",
  encode = "json",
  body = list(
    genus = 'Gagea',
    species = 'pratensis'
  )
)
jsonlite::fromJSON(res$parse())

## End(Not run)
```

---

**Verb PUT**

**HTTP verb info: PUT**

---

**Description**

The PUT method replaces all current representations of the target resource with the request payload.

**The PUT method**

The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload. A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent in a 200 (OK) response. However, there is no guarantee that such a state change will be observable, since the target resource might be acted upon by other user agents in parallel, or might be subject to dynamic processing by the origin server, before any subsequent GET is received. A successful response only implies that the user agent’s intent was achieved at the time of its processing by the origin server.

If the target resource does not have a current representation and the PUT successfully creates one, then the origin server MUST inform the user agent by sending a 201 (Created) response. If the target resource does have a current representation and that representation is successfully modified in accordance with the state of the enclosed representation, then the origin server MUST send either a 200 (OK) or a 204 (No Content) response to indicate successful completion of the request.

See [https://tools.ietf.org/html/rfc7231#section-4.3.4](https://tools.ietf.org/html/rfc7231#section-4.3.4) for further details.

**References**

[https://tools.ietf.org/html/rfc7231#section-4.3.4](https://tools.ietf.org/html/rfc7231#section-4.3.4)
See Also

[crul-package](#)

Other verbs: verb-DELETE, verb-GET, verb-HEAD, verb-PATCH, verb-POST

Examples

```r
## Not run:
x <- HttpClient$new(url = "https://httpbin.org")
x$put(path = 'put', body = list(foo = "bar"))

## End(Not run)
```

---

**writing-options**  
Writing data options

**Examples**

```r
## Not run:
# write to disk
(x <- HttpClient$new(url = "https://httpbin.org"))
f <- tempfile()
res <- x$get("get", disk = f)
res$content # when using write to disk, content is a path
readLines(res$content)
close(file(f))

# streaming response
(x <- HttpClient$new(url = "https://httpbin.org"))
res <- x$get('stream/50', stream = function(x) cat(rawToChar(x)))
res$content # when streaming, content is NULL

## Async
(cc <- Async$new( 
  urls = c( 
    'https://httpbin.org/get?a=5', 
    'https://httpbin.org/get?foo=bar', 
    'https://httpbin.org/get?b=4', 
    'https://httpbin.org/get?stuff=things', 
    'https://httpbin.org/get?b=4&g=7&u=9&z=1' 
  ) 
))
files <- replicate(5, tempfile())
(res <- cc$get(disk = files, verbose = TRUE))
```
lapply(files, readLines)

## Async varied
### disk
f <- tempfile()
g <- tempfile()
req1 <- HttpRequest$new(url = "https://httpbin.org/get")$get(disk = f)
req2 <- HttpRequest$new(url = "https://httpbin.org/post")$post(disk = g)
req3 <- HttpRequest$new(url = "https://httpbin.org/get")$get()
(out <- AsyncVaried$new(req1, req2, req3))
out$request()
out$content()
readLines(f)
readLines(g)
out$parse()
close(file(f))
close(file(g))

### stream - to console
fun <- function(x) print(x)
req1 <- HttpRequest$new(url = "https://httpbin.org/get" )$get(query = list(foo = "bar"), stream = fun)
req2 <- HttpRequest$new(url = "https://httpbin.org/get" )$get(query = list(hello = "world"), stream = fun)
(out <- AsyncVaried$new(req1, req2))
out$request()
out$content()

### stream - to an R object
lst <- list()
fun <- function(x) lst <<- append(lst, list(x))
req1 <- HttpRequest$new(url = "https://httpbin.org/get" )$get(query = list(foo = "bar"), stream = fun)
req2 <- HttpRequest$new(url = "https://httpbin.org/get" )$get(query = list(hello = "world"), stream = fun)
(out <- AsyncVaried$new(req1, req2))
out$request()
out$content()
lst
cat(vapply(lst, function(z) rawToChar(z$content), "", sep = "\n"))

## End(Not run)
Index

* **async**
  Async, 4
  AsyncQueue, 11
  AsyncVaried, 13
  HttpRequest, 38

* **verbs**
  verb-DELETE, 60
  verb-GET, 61
  verb-HEAD, 62
  verb-PATCH, 63
  verb-POST, 64
  verb-PUT, 65

Async, 3, 4, 12, 16, 42
Async(), 3
AsyncQueue, 9, 11, 16, 42
AsyncVaried, 3, 9, 12, 13, 42
AsyncVaried(), 3, 38
auth, 18
auth(), 3, 6, 22, 29, 30, 38, 39

cert_types, 3, 19, 46
cookies, 21, 35
crul (crul-package), 2
crul-options, 22
crul-package, 2, 60, 62–64, 66

crul::AsyncVaried, 11

crul_settings (crul-options), 22
crul_settings(), 3
curl-options, 24
curl::curl_fetch_disk(), 6–8, 30–33,
39–41

curl::curl_fetch_stream(), 6–8, 30–33,
39–41

curl::curl_options(), 6–9, 22, 24, 30–34,
39–42, 52–54

curl::new_handle(), 26
curl_verbose, 25
curl_verbose(), 22

format(), 30–33, 35, 50–55
handle, 26
handle(), 29, 30, 38, 39
hooks, 26, 29, 30, 35
http_headers, 27, 29, 30, 35, 38, 39, 42
HttpClient, 3, 4, 14, 26, 28, 48, 52, 57
HttpClient(), 3, 4, 38, 50
HttpRequest, 3, 9, 12, 16, 38
HttpRequest(), 3, 12, 15
HttpResponse, 5, 14, 19, 20, 28, 43
HttpResponse(), 3, 4, 50

mime::guess_type, 59
mime::mimemap, 20, 43
mock, 4, 47
mock(), 3, 22

ok, 48

Paginator, 50
Paginator(), 3
progress, 29, 30, 38, 39, 57
proxies, 57
proxy (proxies), 57
proxy(), 3, 6, 22, 29, 30, 38, 39

set_auth (crul-options), 22
set_auth(), 3
set_headers (crul-options), 22
set_headers(), 3
set_opts (crul-options), 22
set_opts(), 3
set_proxy (crul-options), 22
set_proxy(), 3
set_verbose (crul-options), 22
timeout (curl-options), 24

upload, 59
upload(), 3
url_build, 59
url_parse (url_build), 59
user-agent (curl-options), 24
utils::txtProgressBar, 51, 52
verb-DELETE, 3, 60
verb-GET, 3, 61
verb-HEAD, 3, 62
verb-PATCH, 3, 63
verb-POST, 3, 64
verb-PUT, 3, 65
verbose (curl-options), 24

writing-options, 35, 42, 66