

# Package ‘crul’

July 10, 2018

**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.haxx.se/libcurl>>).

**Version** 0.6.0

**License** MIT + file LICENSE

**URL** <https://github.com/ropensci/crul> (devel)  
<https://ropensci.github.io/http-testing-book/> (user manual)

**BugReports** <https://github.com/ropensci/crul/issues>

**Language** en-US

**Imports** curl (>= 3.1), R6 (>= 2.2.0), urltools (>= 1.6.0), httpcode (>= 0.2.0), mime

**Suggests** testthat, fauxpas (>= 0.1.0), webmockr (>= 0.1.0), knitr, jsonlite

**VignetteBuilder** knitr

**RoxygenNote** 6.0.1

**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](mailto:myrmecocystus@gmail.com)>

**Repository** CRAN

**Date/Publication** 2018-07-10 20:00:03 UTC

## R topics documented:

crul-package . . . . .	2
Async . . . . .	3
AsyncVaried . . . . .	5
auth . . . . .	8
cookies . . . . .	9
crul-options . . . . .	10
curl-options . . . . .	12
delete-requests . . . . .	13
handle . . . . .	13
http-headers . . . . .	14
HttpClient . . . . .	15
HttpRequest . . . . .	17
HttpResponse . . . . .	19
HttpStubbedResponse . . . . .	20
mock . . . . .	21
ok . . . . .	22
Paginator . . . . .	23
post-requests . . . . .	25
progress . . . . .	26
proxies . . . . .	27
upload . . . . .	28
url_build . . . . .	28
writing-options . . . . .	29
<b>Index</b>	<b>32</b>

---

crul-package

**HTTP R client**

---

### 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

### HTTP conditions

We use `fauxpas` if you have it installed for handling HTTP conditions but if it's not installed we use `httpcode`

### Mocking

Mocking HTTP requests is supported via the `webmockr` package. See [mock](#) for guidance, and <https://ropensci.github.io/http-testing-book/>

### Caching

Caching HTTP requests is supported via the `vcr` package. See <https://ropensci.github.io/http-testing-book/>

### Author(s)

Scott Chamberlain <[myrmecocystus@gmail.com](mailto:myrmecocystus@gmail.com)>

---

Async

*Simple async client*

---

### Description

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

### Arguments

`urls` (character) one or more URLs (required)

### Details

#### Methods

`get(path, query, disk, stream, ...)` make async GET requests for all URLs

`post(path, query, body, encode, disk, stream, ...)` make async POST requests for all URLs

`put(path, query, body, encode, disk, stream, ...)` make async PUT requests for all URLs

`patch(path, query, body, encode, disk, stream, ...)` make async PATCH requests for all URLs

`delete(path, query, body, encode, disk, stream, ...)` make async DELETE requests for all URLs

`head(path, ...)` make async HEAD requests for all URLs

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 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

**See Also**

Other async: [AsyncVaried](#)

**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]]
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"))
```

```

# using auth with async
dd <- Async$new(urls = rep('https://httpbin.org/basic-auth/user/passwd', 3))
res <- dd$get(auth = auth(user = "user", pwd = "passwd"))
res
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.
urls <- c("http://stuffthings.gvb", "https://foo.com",
         "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)

```

---

 AsyncVaried

*Async client for different request types*


---

## Description

Async client for different request types

## Arguments

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

## Details

### Methods

`request()` Execute asynchronous requests - returns: nothing, responses stored inside object, though will print messages if you choose verbose output

`requests()` list requests - returns: a list of `HttpRequest` objects, empty list before requests made

`responses()` list responses - returns: a list of `HttpResponse` objects, empty list before requests made

`parse(encoding = "UTF-8")` parse content - returns: character vector, empty character vector before requests made

`status_code()` (integer) HTTP status codes - returns: numeric vector, empty numeric vector before requests made

`status()` (list) HTTP status objects - returns: a list of `http_code` objects, empty list before requests made

content() raw content - returns: raw list, empty list before requests made

times() curl request times - returns: list of named numeric vectors, empty list before requests made

### Value

An object of class AsyncVaried with variables and methods. 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 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

### See Also

Other async: [Async](#)

### 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()

# 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(
  HttpRequest$new(url = url, auth = auth)$get(),
  HttpRequest$new(url = url, auth = auth)$get(query = list(a=5)),
  HttpRequest$new(url = url, auth = auth)$get(query = list(b=3))
)
out <- AsyncVaried$new(.list = reqlist)
out$request()
out$status()

```

```
out$parse()

# failure behavior
## e.g. when a URL doesn't exist, a timeout, etc.
reqlist <- list(
  HttpRequest$new(url = "http://stuffthings.gvb")$get(),
  HttpRequest$new(url = "https://httpbin.org")$head(),
  HttpRequest$new(url = "https://httpbin.org",
    opts = list(timeout_ms = 10))$head()
)
(tmp <- AsyncVaried$new(.list = reqlist))
tmp$request()
tmp$responses()
tmp$parse("UTF-8")

## End(Not run)
```

---

auth

*Authentication*

---

## 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 `httr::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)

```

---

cookies

*Working with cookies*


---

**Description**

Working with cookies

**Examples**

```

## 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(...)
```

```
set_proxy(x)
```

```
set_auth(x)
```

```
set_headers(...)
```

```
crul_settings(reset = FALSE)
```

**Arguments**

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

**Details**

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

**Examples**

```
# get settings
crul_settings()

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

## End(Not run)

# 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(verbose = TRUE)
HttpClient$new('https://httpbin.org')$get('get')

## End(Not run)

# reset
crul_settings(TRUE)
crul_settings()
```

```

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

## AsyncVaried
set_opts(verbose = 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

```

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

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

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

HttpClient$new(url = url,
  opts = list(verbose = TRUE, useragent = "hello world")
)$get('get')

## End(Not run)

```

---

delete-requests	<i>HTTP DELETE requests</i>
-----------------	-----------------------------

---

### Description

HTTP DELETE requests

### Examples

```

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

## 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)

```

---

handle	<i>Make a handle</i>
--------	----------------------

---

### 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")

# 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)
```

---

 http-headers

 Working with HTTP headers
 

---

**Description**

Working with HTTP headers

**Examples**

```
## 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

HTTP client

## Arguments

url	(character) A url. One of url or handle required.
opts	(list) curl options, a named list. See <a href="#">curl_options</a> for available curl options
proxies	an object of class proxy, as returned from the <a href="#">proxy</a> function. Supports one proxy for now
auth	result of a call to the <a href="#">auth</a> function, e.g. auth(user = "foo", pwd = "bar")
headers	(list) a named list of headers
handle	A handle, see <a href="#">handle</a>
progress	a function with logic for printing a progress bar for an HTTP request, ultimately passed down to <b>curl</b> . only supports htr::progress() for now

## Details

### Methods

get(path, query, disk, stream, ...) Make a GET request  
 post(path, query, body, disk, stream, ...) Make a POST request  
 put(path, query, body, disk, stream, ...) Make a PUT request  
 patch(path, query, body, disk, stream, ...) Make a PATCH request  
 delete(path, query, body, disk, stream, ...) Make a DELETE request  
 head(path, query, ...) Make a HEAD request

Possible parameters (not all are allowed in each HTTP verb):

- path - URL path, appended to the base URL
- query - 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

### 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 `Http` slot.

### Note

a little quark about `curl` 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

[post-requests](#), [delete-requests](#), [http-headers](#), [writing-options](#), [cookies](#)

### Examples

```
## 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

# 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'))

## End(Not run)

```

---

HttpRequest

*HTTP request object*


---

## Description

HTTP request object

## Arguments

`url` (character) A url. One of `url` or `handle` required.

`opts` (list) curl options, a named list. See [curl\\_options](#) for available curl options

proxies	an object of class proxy, as returned from the <a href="#">proxy</a> function. Supports one proxy for now
auth	result of a call to the <a href="#">auth</a> function, e.g. <code>auth(user = "foo", pwd = "bar")</code>
headers	(list) a named list of headers
handle	A handle, see <a href="#">handle</a>
progress	a function with logic for printing a progress bar for an HTTP request, ultimately passed down to <b>curl</b> . only supports <code>httr::progress()</code> for now

### 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.

### Methods

`get(path, query, disk, stream, ...)` Define a GET request

`post(path, query, body, disk, stream, ...)` Define a POST request

`put(path, query, body, disk, stream, ...)` Define a PUT request

`patch(path, query, body, disk, stream, ...)` Define a PATCH request

`delete(path, query, body, disk, stream, ...)` Define a DELETE request

`head(path, ...)` Define a HEAD request

`method()` Get the HTTP method (if defined) - returns character string

See [HttpClient\(\)](#) for information on parameters.

### See Also

[post-requests](#), [delete-requests](#), [http-headers](#), [writing-options](#)

### 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)
```

---

HttpResponse	<i>Base response object</i>
--------------	-----------------------------

---

### Description

Base response object

### Arguments

url	(character) A url, required
opts	(list) curl options
handle	A handle
method	(character) HTTP method
status_code	(integer) status code
request_headers	(list) request headers, named list
response_headers	(list) response headers, named list
modified	(character) modified date
times	(vector) named vector
content	(raw) raw binary content response
request	request object, with all details

### Details

#### Methods

parse() Parse the raw response content to text

success() Was status code less than or equal to 201. returns boolean

status\_http() Get HTTP status code, message, and explanation

raise\_for\_status() Check HTTP status and stop with appropriate HTTP error code and message if >= 300. - If you have fauxpas installed we use that, otherwise use **httplibcode**

**Examples**

```
## 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('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()$explanation
res$success()

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)
```

---

HttpStubbedResponse    *stubbed response object*

---

**Description**

stubbed response object

**Arguments**

url	(character) A url
opts	(list) curl options
handle	A handle

**Details****Methods**

parse() Parse the raw response content to text  
 success() Was status code less than or equal to 201. returns boolean  
 status\_http() Get HTTP status code, message, and explanation  
 raise\_for\_status() Check HTTP status and stop with appropriate HTTP error code and message if  $\geq 300$ . - If you have fauxpas installed we use that, otherwise use **httpcode**

## Examples

```
(x <- HttpStubbedResponse$new(method = "get", url = "https://httpbin.org"))
x$url
x$method
```

---

mock

*Mocking HTTP requests*

---

## Description

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

```
## Not run:

if (interactive()) {
  # load webmockr
  library(webmockr)
  library(cru1)

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

  # turn on mocking
  cru1::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

```
ok(x, status = 200L, info = TRUE, ...)
```

## Arguments

x	either a URL as a character string, or an object of class <a href="#">HttpClient</a>
status	(integer) an HTTP status code, must be an integer. By default this is 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
...	args passed on to <a href="#">HttpClient</a>

## Details

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

## Value

a single boolean, if TRUE the URL is up and okay, if FALSE it is down.

## Examples

```

## Not run:
# 200
ok("https://google.com")
# 200
ok("https://httpbin.org/status/200")
# 404
ok("https://httpbin.org/status/404")

```

```

# doesn't exist
ok("https://stuff.bar")
# doesn't exist
ok("stuff")

# with HttpClient
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

## Arguments

<code>client</code>	an object of class <code>HttpClient</code> , from a call to <a href="#">HttpClient</a>
<code>by</code>	(character) how to paginate. Only 'query_params' supported for now. In the future will support 'link_headers' and 'cursor'. See Details.
<code>limit_param</code>	(character) the name of the limit parameter. Default: limit
<code>offset_param</code>	(character) the name of the offset parameter. Default: offset
<code>limit</code>	(numeric/integer) the maximum records wanted
<code>limit_chunk</code>	(numeric/integer) the number by which to chunk requests, e.g., 10 would be each request gets 10 records

## Details

### Methods

`get(path, query, ...)` make a paginated GET request

`post(path, query, body, encode, ...)` make a paginated POST request

`put(path, query, body, encode, ...)` make a paginated PUT request

`patch(path, query, body, encode, ...)` make a paginated PATCH request

`delete(path, query, body, encode, ...)` make a paginated DELETE request

`head(path, ...)` make a paginated HEAD request - not sure if this makes any sense or not yet

`responses()` list responses - returns: a list of `HttpResponse` objects, empty list before requests made

`parse(encoding = "UTF-8")` parse content - returns: character vector, empty character vector before requests made

`status_code()` (integer) HTTP status codes - returns: numeric vector, empty numeric vector before requests made

`status()` (list) HTTP status objects - returns: a list of `http_code` objects, empty list before requests made

`content()` raw content - returns: raw list, empty list before requests made

`times()` curl request times - returns: list of named numeric vectors, empty list before requests made

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:

- `query_params`: the most common way, 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.

Supported later:

- `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

### Examples

```
## Not run:
(cli <- HttpClient$new(url = "http://api.crossref.org"))
cc <- Paginator$new(client = cli, limit_param = "rows",
  offset_param = "offset", limit = 50, limit_chunk = 10)
cc
cc$get('works')
cc
cc$responses()
cc$status()
cc$status_code()
cc$times()
cc$content()
cc$parse()
lapply(cc$parse(), jsonlite::fromJSON)

## End(Not run)
```



**Description**

HTTP POST/PUT/PATCH requests

**Examples**

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

# POST requests
## a list
(res_post <- x$post('post', body = list(hello = "world"), verbose = TRUE))

## a string
(res_post <- x$post('post', body = "hello world", verbose = TRUE))

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

## form requests
(cli <- HttpClient$new(
  url = "http://apps.kew.org/wcsp/advsearch.do"
))
cli$post(
  encode = "form",
  body = list(
    page = 'advancedSearch',
    genus = 'Gagea',
    species = 'pratensis',
    selectedLevel = 'cont'
  )
)

(x <- HttpClient$new(url = "https://httpbin.org"))
res <- x$post("post",
  encode = "json",
  body = list(
    genus = 'Gagea',
    species = 'pratensis'
  )
)
jsonlite::fromJSON(res$parse())

# PUT requests
(x <- HttpClient$new(url = "https://httpbin.org"))
(res <- x$put(path = "put",
  encode = "json",
```

```

    body = list(
      genus = 'Gagea',
      species = 'pratensis'
    )
  ))
  jsonlite::fromJSON(res$parse("UTF-8"))

res <- x$put("put", body = "foo bar")
jsonlite::fromJSON(res$parse("UTF-8"))

# PATCH requests
(x <- HttpClient$new(url = "https://httpbin.org"))
(res <- x$patch(path = "patch",
  encode = "json",
  body = list(
    genus = 'Gagea',
    species = 'pratensis'
  )
))
jsonlite::fromJSON(res$parse("UTF-8"))

res <- x$patch("patch", body = "foo bar")
jsonlite::fromJSON(res$parse("UTF-8"))

# Upload files
## image
path <- file.path(Sys.getenv("R_DOC_DIR"), "html/logo.jpg")
(x <- HttpClient$new(url = "https://httpbin.org"))
res <- x$post(path = "post", body = list(y = upload(path)))
res$content

## text file, in a list
(x <- HttpClient$new(url = "https://httpbin.org"))
file <- upload(system.file("CITATION"))
res <- x$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)

```

---

progress

*progress bars*

---

## 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

---

proxies

*proxy options*

---

**Description**

proxy options

**Usage**

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

**Arguments**

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

**Details**

See <http://proxylist.hidemyass.com/> 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(verbose = TRUE)

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

```

---

upload

*upload file*

---

### 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

---

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
path         (character) a path
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[]")
```

---

writing-options

*Writing data options*

---

**Description**

Writing data options

**Examples**

```
## 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)
close(file(f))
close(file(g))

### stream - to console
fun <- function(x) cat(rawToChar(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 <- c()
fun <- function(x) lst <- c(lst, 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()  
lst  
cat(rawToChar(lst))  
  
## End(Not run)
```

# Index

## \*Topic **datasets**

- Async, [3](#)
  - AsyncVaried, [5](#)
  - HttpClient, [15](#)
  - HttpRequest, [17](#)
  - HttpResponse, [19](#)
  - HttpStubbedResponse, [20](#)
  - Paginator, [23](#)
- Async, [3](#), [6](#)
- Async(), [2](#)
- AsyncVaried, [4](#), [5](#)
- AsyncVaried(), [2](#), [18](#)
- auth, [8](#), [15](#), [18](#)
- auth(), [2](#), [11](#)
- cookies, [9](#), [16](#)
- curl (curl-package), [2](#)
- curl-options, [10](#)
- curl-package, [2](#)
- curl\_settings (curl-options), [10](#)
- curl-options, [12](#)
- curl::curl\_fetch\_disk(), [16](#)
- curl::curl\_fetch\_stream(), [16](#)
- curl::curl\_options(), [11](#), [12](#), [16](#)
- curl::new\_handle(), [14](#)
- curl\_options, [15](#), [17](#)
- delete-requests, [13](#), [16](#), [18](#)
- handle, [13](#), [15](#), [18](#)
- http-headers, [14](#), [16](#), [18](#)
- HttpClient, [15](#), [22](#), [23](#), [27](#)
- HttpClient(), [2](#), [3](#), [18](#), [24](#)
- HttpRequest, [17](#)
- HttpRequest(), [2](#), [5](#)
- HttpResponse, [19](#)
- HttpResponse(), [2](#), [4](#), [24](#)
- HttpStubbedResponse, [20](#)
- mime::guess\_type, [28](#)
- mock, [3](#), [21](#)
- mock(), [2](#), [11](#)
- ok, [22](#)
- Paginator, [23](#)
- Paginator(), [2](#)
- post-requests, [16](#), [18](#), [25](#)
- progress, [26](#)
- proxies, [27](#)
- proxy, [15](#), [18](#)
- proxy (proxies), [27](#)
- proxy(), [2](#), [11](#)
- set\_auth (curl-options), [10](#)
- set\_headers (curl-options), [10](#)
- set\_opts (curl-options), [10](#)
- set\_proxy (curl-options), [10](#)
- timeout (curl-options), [12](#)
- upload, [28](#)
- url\_build, [28](#)
- url\_parse (url\_build), [28](#)
- user-agent (curl-options), [12](#)
- verbose (curl-options), [12](#)
- writing-options, [16](#), [18](#), [29](#)