Package 'curl'

September 24, 2019

Type Package

Title A Modern and Flexible Web Client for R

Version 4.2

Description The curl() and curl_download() functions provide highly configurable drop-in replacements for base url() and download.file() with better performance, support for encryption (https, ftps), gzip compression, authentication, and other 'libcurl' goodies. The core of the package implements a framework for performing fully customized requests where data can be processed either in memory, on disk, or streaming via the callback or connection interfaces. Some knowledge of 'libcurl' is recommended; for a more-user-friendly web client see the 'httr' package which builds on this package with http specific tools and logic.

License MIT + file LICENSE

SystemRequirements libcurl: libcurl-devel (rpm) or libcurl4-openssl-dev (deb).

URL https://jeroen.cran.dev/curl (docs)
https://github.com/jeroen/curl#readme (devel)
https://curl.haxx.se/libcurl/ (upstream)

BugReports https://github.com/jeroen/curl/issues

Suggests askpass, spelling, testthat (>= 1.0.0), knitr, jsonlite, rmarkdown, magrittr, httpuv (>= 1.4.4), webutils

VignetteBuilder knitr

Depends R (>= 3.0.0)

LazyData true

RoxygenNote 6.1.1

Encoding UTF-8

Language en-US

NeedsCompilation yes

Author Jeroen Ooms [aut, cre] (<https://orcid.org/0000-0002-4035-0289>), Hadley Wickham [ctb], RStudio [cph]

Maintainer Jeroen Ooms <jeroen@berkeley.edu>

Repository CRAN

Date/Publication 2019-09-24 12:20:02 UTC

R topics documented:

`curl` *Curl connection interface*

Description

Drop-in replacement for base `url` that supports https, ftps, gzip, deflate, etc. Default behavior is identical to `url`, but request can be fully configured by passing a custom `handle`.

Usage

```
curl(url = "http://httpbin.org/get", open = "",
  handle = new_handle())
```

Arguments

`url`	character string. See examples.
`open`	character string. How to open the connection if it should be opened initially. Currently only "r" and "rb" are supported.
`handle`	a curl handle object

Details

As of version 2.3 curl connections support `open(con,blocking = FALSE)`. In this case `readBin` and `readLines` will return immediately with data that is available without waiting. For such non-blocking connections the caller needs to call `isIncomplete` to check if the download has completed yet.

Examples

```
## Not run:
con <- curl("https://httpbin.org/get")
readLines(con)

# Auto-opened connections can be recycled
open(con, "rb")
bin <- readBin(con, raw(), 999)
close(con)
rawToChar(bin)

# HTTP error
curl("https://httpbin.org/status/418", "r")

# Follow redirects
readLines(curl("https://httpbin.org/redirect/3"))

# Error after redirect
curl("https://httpbin.org/redirect-to?url=http://httpbin.org/status/418", "r")

# Auto decompress Accept-Encoding: gzip / deflate (rfc2616 #14.3)
readLines(curl("http://httpbin.org/gzip"))
readLines(curl("http://httpbin.org/deflate"))

# Binary support
buf <- readBin(curl("http://httpbin.org/bytes/98765", "rb"), raw(), 1e5)
length(buf)

# Read file from disk
test <- paste0("file://", system.file("DESCRIPTION"))
readLines(curl(test))

# Other protocols
read.csv(curl("ftp://cran.r-project.org/pub/R/CRAN_mirrors.csv"))
readLines(curl("ftps://test.rebex.net:990/readme.txt"))
readLines(curl("gopher://quux.org/1"))

# Streaming data
con <- curl("http://jeroen.github.io/data/diamonds.json", "r")
while(length(x <- readLines(con, n = 5))){
  print(x)
}

# Stream large dataset over https with gzip
library(jsonlite)
```

```
con <- gzcon(curl("https://jeroen.github.io/data/nycflights13.json.gz"))
nycflights <- stream_in(con)

## End(Not run)
```

---

curl_download *Download file to disk*

Description

Libcurl implementation of C_download (the "internal" download method) with added support for https, ftps, gzip, etc. Default behavior is identical to download.file, but request can be fully configured by passing a custom handle.

Usage

```
curl_download(url, destfile, quiet = TRUE, mode = "wb",
  handle = new_handle())
```

Arguments

url	A character string naming the URL of a resource to be downloaded.
destfile	A character string with the name where the downloaded file is saved. Tilde-expansion is performed.
quiet	If TRUE, suppress status messages (if any), and the progress bar.
mode	A character string specifying the mode with which to write the file. Useful values are "w", "wb" (binary), "a" (append) and "ab".
handle	a curl handle object

Details

The main difference between curl_download and curl_fetch_disk is that curl_download checks the http status code before starting the download, and raises an error when status is non-successful. The behavior of curl_fetch_disk on the other hand is to proceed as normal and write the error page to disk in case of a non success response.

Value

Path of downloaded file (invisibly).

Examples

```
# Download large file
## Not run:
url <- "http://www2.census.gov/acs2011_5yr/pums/csv_pus.zip"
tmp <- tempfile()
curl_download(url, tmp)

## End(Not run)
```

---

curl_echo *Echo Service*

Description

This function is only for testing purposes. It starts a local httpuv server to echo the request body and content type in the response.

Usage

```
curl_echo(handle, port = 9359, progress = interactive(), file = NULL)
```

Arguments

handle	a curl handle object
port	the port number on which to run httpuv server
progress	show progress meter during http transfer
file	path or connection to write body. Default returns body as raw vector.

Examples

```
h <- new_handle(url = 'https://httpbin.org/post')
handle_setform(h, foo = "blabla", bar = charToRaw("test"),
  myfile = form_file(system.file("DESCRIPTION"), "text/description"))

# Echo the POST request data
formdata <- curl_echo(h)

# Show the multipart body
cat(rawToChar(formdata$body))

# Parse multipart
webutils::parse_http(formdata$body, formdata$content_type)
```

curl_escape *URL encoding*

Description

Escape all special characters (i.e. everything except for a-z, A-Z, 0-9, '-', '.', '_' or '~') for use in URLs.

Usage

```
curl_escape(url)

curl_unescape(url)
```

Arguments

url	A character vector (typically containing urls or parameters) to be encoded/decoded

Examples

```
# Escape strings
out <- curl_escape("foo = bar + 5")
curl_unescape(out)

# All non-ascii characters are encoded
mu <- "\u00b5"
curl_escape(mu)
curl_unescape(curl_escape(mu))
```

curl_fetch_memory *Fetch the contents of a URL*

Description

Low-level bindings to write data from a URL into memory, disk or a callback function. These are mainly intended for `httr`, most users will be better off using the `curl` or `curl_download` function, or the http specific wrappers in the `httr` package.

Usage

```
curl_fetch_memory(url, handle = new_handle())

curl_fetch_disk(url, path, handle = new_handle())

curl_fetch_stream(url, fun, handle = new_handle())
```

```
curl_fetch_multi(url, done = NULL, fail = NULL, pool = NULL,
  data = NULL, handle = new_handle())

curl_fetch_echo(url, handle = new_handle())
```

Arguments

url	A character string naming the URL of a resource to be downloaded.
handle	a curl handle object
path	Path to save results
fun	Callback function. Should have one argument, which will be a raw vector.
done	callback function for completed request. Single argument with response data in same structure as curl_fetch_memory.
fail	callback function called on failed request. Argument contains error message.
pool	a multi handle created by new_pool. Default uses a global pool.
data	(advanced) callback function, file path, or connection object for writing incoming data. This callback should only be used for *streaming* applications, where small pieces of incoming data get written before the request has completed. The signature for the callback function is `write(data,final = FALSE)`. If set to `NULL` the entire response gets buffered internally and returned by in the done callback (which is usually what you want).

Details

The curl_fetch functions automatically raise an error upon protocol problems (network, disk, ssl) but do not implement application logic. For example for you need to check the status code of http requests yourself in the response, and deal with it accordingly.

Both `curl_fetch_memory` and `curl_fetch_disk` have a blocking and non-blocking C implementation. The latter is slightly slower but allows for interrupting the download prematurely (using e.g. CTRL+C or ESC). Interrupting is enabled when R runs in interactive mode or when `getOption("curl_interrupt") == TRUE`.

The `curl_fetch_multi` function is the asynchronous equivalent of `curl_fetch_memory`. It wraps `multi_add` to schedule requests which are executed concurrently when calling `multi_run`. For each successful request the done callback is triggered with response data. For failed requests (when `curl_fetch_memory` would raise an error), the `fail` function is triggered with the error message.

Examples

```
# Load in memory
res <- curl_fetch_memory("http://httpbin.org/cookies/set?foo=123&bar=ftw")
res$content

# Save to disk
res <- curl_fetch_disk("http://httpbin.org/stream/10", tempfile())
res$content
readLines(res$content)
```

```
# Stream with callback
res <- curl_fetch_stream("http://www.httpbin.org/drip?duration=5&numbytes=15&code=200", function(x){
  cat(rawToChar(x))
})

# Async API
data <- list()
success <- function(res){
  cat("Request done! Status:", res$status, "\n")
  data <<- c(data, list(res))
}
failure <- function(msg){
  cat("Oh noes! Request failed!", msg, "\n")
}
curl_fetch_multi("http://httpbin.org/get", success, failure)
curl_fetch_multi("http://httpbin.org/status/418", success, failure)
curl_fetch_multi("https://urldoesnotexist.xyz", success, failure)
multi_run()
str(data)
```

---

curl_options *List curl version and options.*

Description

curl_version() shows the versions of libcurl, libssl and zlib and supported protocols. curl_options() lists all options available in the current version of libcurl. The dataset curl_symbols lists all symbols (including options) provides more information about the symbols, including when support was added/removed from libcurl.

Usage

```
curl_options(filter = "")

curl_symbols(filter = "")

curl_version()
```

Arguments

filter	string: only return options with string in name

Examples

```
# Available options
curl_options()

# List proxy options
curl_options("proxy")
```

```
# Symbol table
curl_symbols("proxy")
# Curl/ssl version info
curl_version()
```

`curl_upload` *Upload a File*

Description

Upload a file to an `http://`, `ftp://`, or `sftp://` (ssh) server. Uploading to HTTP means performing an `HTTP PUT` on that URL. Be aware that sftp is only available for libcurl clients built with libssh2.

Usage

```
curl_upload(file, url, verbose = TRUE, reuse = TRUE, ...)
```

Arguments

`file`	connection object or path to an existing file on disk
`url`	where to upload, should start with e.g. `ftp://`
`verbose`	emit some progress output
`reuse`	try to keep alive and recycle connections when possible
`...`	other arguments passed to `handle_setopt`, for example a `username` and `password`.

Examples

```
# Upload package to winbuilder:
curl_upload('mypkg_1.3.tar.gz', 'ftp://win-builder.r-project.org/R-devel/')
```

`file_writer` *File Writer*

Description

Generates a closure that writes binary (raw) data to a file.

Usage

```
file_writer(path)
```

Arguments

path	file name or path on disk

Details

The writer function automatically opens the file on the first write and closes when it goes out of scope, or explicitly by setting close = TRUE. This can be used for the data callback in multi_add() or curl_fetch_multi().

Value

Function with signature writer(data = raw(),close = FALSE)

Examples

```
# Doesn't open yet
tmp <- tempfile()
writer <- file_writer(tmp)

# Now it opens
writer(charToRaw("Hello!\n"))
writer(charToRaw("How are you?\n"))

# Close it!
writer(charToRaw("All done!\n"), close = TRUE)

# Check it worked
readLines(tmp)
```

---

handle *Create and configure a curl handle*

Description

Handles are the work horses of libcurl. A handle is used to configure a request with custom options, headers and payload. Once the handle has been set up, it can be passed to any of the download functions such as curl ,curl_download or curl_fetch_memory. The handle will maintain state in between requests, including keep-alive connections, cookies and settings.

Usage

```
new_handle(...)

handle_setopt(handle, ..., .list = list())

handle_setheaders(handle, ..., .list = list())

handle_getheaders(handle)
```

```
handle_getcustom(handle)

handle_setform(handle, ..., .list = list())

handle_reset(handle)

handle_data(handle)
```

Arguments

`...`	named options / headers to be set in the handle. To send a file, see `form_file`. To list all allowed options, see `curl_options`
`handle`	Handle to modify
`.list`	A named list of options. This is useful if you've created a list of options elsewhere, avoiding the use of `do.call()`.

Details

Use `new_handle()` to create a new clean curl handle that can be configured with custom options and headers. Note that `handle_setopt` appends or overrides options in the handle, whereas `handle_setheaders` replaces the entire set of headers with the new ones. The `handle_reset` function resets only options/headers/forms in the handle. It does not affect active connections, cookies or response data from previous requests. The safest way to perform multiple independent requests is by using a separate handle for each request. There is very little performance overhead in creating handles.

Value

A handle object (external pointer to the underlying curl handle). All functions modify the handle in place but also return the handle so you can create a pipeline of operations.

See Also

Other handles: `handle_cookies`

Examples

```
h <- new_handle()
handle_setopt(h, customrequest = "PUT")
handle_setform(h, a = "1", b = "2")
r <- curl_fetch_memory("http://httpbin.org/put", h)
cat(rawToChar(r$content))

# Or use the list form
h <- new_handle()
handle_setopt(h, .list = list(customrequest = "PUT"))
handle_setform(h, .list = list(a = "1", b = "2"))
r <- curl_fetch_memory("http://httpbin.org/put", h)
cat(rawToChar(r$content))
```

handle_cookies *Extract cookies from a handle*

Description

The handle_cookies function returns a data frame with 7 columns as specified in the netscape cookie file format.

Usage

```
handle_cookies(handle)
```

Arguments

handle	a curl handle object

See Also

Other handles: handle

Examples

```
h <- new_handle()
handle_cookies(h)

# Server sets cookies
req <- curl_fetch_memory("http://httpbin.org/cookies/set?foo=123&bar=ftw", handle = h)
handle_cookies(h)

# Server deletes cookies
req <- curl_fetch_memory("http://httpbin.org/cookies/delete?foo", handle = h)
handle_cookies(h)

# Cookies will survive a reset!
handle_reset(h)
handle_cookies(h)
```

ie_proxy *Internet Explorer proxy settings*

Description

Lookup and mimic the system proxy settings on Windows as set by Internet Explorer. This can be used to configure curl to use the same proxy server.

Usage

```
ie_proxy_info()

ie_get_proxy_for_url(target_url = "http://www.google.com")
```

Arguments

target_url	url with host for which to lookup the proxy server

Details

The ie_proxy_info function looks up your current proxy settings as configured in IE under "Internet Options" > "Tab: Connections" > "LAN Settings". The ie_get_proxy_for_url determines if and which proxy should be used to connect to a particular URL. If your settings have an "automatic configuration script" this involves downloading and executing a PAC file, which can take a while.

---

multi *Async Multi Download*

Description

AJAX style concurrent requests, possibly using HTTP/2 multiplexing. Results are only available via callback functions. Advanced use only!

Usage

```
multi_add(handle, done = NULL, fail = NULL, data = NULL,
  pool = NULL)

multi_run(timeout = Inf, poll = FALSE, pool = NULL)

multi_set(total_con = 50, host_con = 6, multiplex = TRUE,
  pool = NULL)

multi_list(pool = NULL)

multi_cancel(handle)

new_pool(total_con = 100, host_con = 6, multiplex = TRUE)

multi_fdset(pool = NULL)
```

Arguments

handle	a curl handle with preconfigured url option.
done	callback function for completed request. Single argument with response data in same structure as curl_fetch_memory.

`fail`	callback function called on failed request. Argument contains error message.
`data`	(advanced) callback function, file path, or connection object for writing incoming data. This callback should only be used for *streaming* applications, where small pieces of incoming data get written before the request has completed. The signature for the callback function is `write(data,final = FALSE)`. If set to `NULL` the entire response gets buffered internally and returned by in the `done` callback (which is usually what you want).
`pool`	a multi handle created by new_pool. Default uses a global pool.
`timeout`	max time in seconds to wait for results. Use `0` to poll for results without waiting at all.
`poll`	If `TRUE` then return immediately after any of the requests has completed. May also be an integer in which case it returns after n requests have completed.
`total_con`	max total concurrent connections.
`host_con`	max concurrent connections per host.
`multiplex`	enable HTTP/2 multiplexing if supported by host and client.

Details

Requests are created in the usual way using a curl handle and added to the scheduler with multi_add. This function returns immediately and does not perform the request yet. The user needs to call multi_run which performs all scheduled requests concurrently. It returns when all requests have completed, or case of a `timeout` or `SIGINT` (e.g. if the user presses `ESC` or `CTRL+C` in the console). In case of the latter, simply call multi_run again to resume pending requests.

When the request succeeded, the `done` callback gets triggered with the response data. The structure if this data is identical to curl_fetch_memory. When the request fails, the `fail` callback is triggered with an error message. Note that failure here means something went wrong in performing the request such as a connection failure, it does not check the http status code. Just like curl_fetch_memory, the user has to implement application logic.

Raising an error within a callback function stops execution of that function but does not affect other requests.

A single handle cannot be used for multiple simultaneous requests. However it is possible to add new requests to a pool while it is running, so you can re-use a handle within the callback of a request from that same handle. It is up to the user to make sure the same handle is not used in concurrent requests.

The multi_cancel function can be used to cancel a pending request. It has no effect if the request was already completed or canceled.

The multi_fdset function returns the file descriptors curl is polling currently, and also a timeout parameter, the number of milliseconds an application should wait (at most) before proceeding. It is equivalent to the `curl_multi_fdset` and `curl_multi_timeout` calls. It is handy for applications that is expecting input (or writing output) through both curl, and other file descriptors.

Examples

```
results <- list()
success <- function(x){
```

```
  results <<- append(results, list(x))
}
failure <- function(str){
  cat(paste("Failed request:", str), file = stderr())
}
# This handle will take longest (3sec)
h1 <- new_handle(url = "https://eu.httpbin.org/delay/3")
multi_add(h1, done = success, fail = failure)

# This handle writes data to a file
con <- file("output.txt")
h2 <- new_handle(url = "https://eu.httpbin.org/post", postfields = "bla bla")
multi_add(h2, done = success, fail = failure, data = con)

# This handle raises an error
h3 <- new_handle(url = "https://urldoesnotexist.xyz")
multi_add(h3, done = success, fail = failure)

# Actually perform the requests
multi_run(timeout = 2)
multi_run()

# Check the file
readLines("output.txt")
unlink("output.txt")
```

`multipart` *POST files or data*

Description

Build multipart form data elements. The `form_file` function uploads a file. The `form_data` function allows for posting a string or raw vector with a custom content-type.

Usage

```
form_file(path, type = NULL)

form_data(value, type = NULL)
```

Arguments

`path`	a string with a path to an existing file on disk
`type`	MIME content-type of the file.
`value`	a character or raw vector to post

nslookup *Lookup a hostname*

Description

The nslookup function is similar to nsl but works on all platforms and can resolve ipv6 addresses if supported by the OS. Default behavior raises an error if lookup fails.

Usage

```
nslookup(host, ipv4_only = FALSE, multiple = FALSE, error = TRUE)

has_internet()
```

Arguments

host	a string with a hostname
ipv4_only	always return ipv4 address. Set to 'FALSE' to allow for ipv6 as well.
multiple	returns multiple ip addresses if possible
error	raise an error for failed DNS lookup. Otherwise returns NULL.

Details

The has_internet function tests for internet connectivity by performing a dns lookup. If a proxy server is detected, it will also check for connectivity by connecting via the proxy.

Examples

```
# Should always work if we are online
nslookup("www.r-project.org")

# If your OS supports IPv6
nslookup("ipv6.test-ipv6.com", ipv4_only = FALSE, error = FALSE)
```

parse_date *Parse date/time*

Description

Can be used to parse dates appearing in http response headers such as Expires or Last-Modified. Automatically recognizes most common formats. If the format is known, strptime might be easier.

Usage

```
parse_date(datestring)
```

Arguments

datestring	a string consisting of a timestamp

Examples

```
# Parse dates in many formats
parse_date("Sunday, 06-Nov-94 08:49:37 GMT")
parse_date("06 Nov 1994 08:49:37")
parse_date("20040911 +0200")
```

---

parse_headers *Parse response headers*

Description

Parse response header data as returned by curl_fetch, either as a set of strings or into a named list.

Usage

```
parse_headers(txt, multiple = FALSE)

parse_headers_list(txt)
```

Arguments

txt	raw or character vector with the header data
multiple	parse multiple sets of headers separated by a blank line. See details.

Details

The parse_headers_list function parses the headers into a normalized (lowercase field names, trimmed whitespace) named list.

If a request has followed redirects, the data can contain multiple sets of headers. When multiple = TRUE, the function returns a list with the response headers for each request. By default it only returns the headers of the final request.

Examples

```
req <- curl_fetch_memory("https://httpbin.org/redirect/3")
parse_headers(req$headers)
parse_headers(req$headers, multiple = TRUE)

# Parse into named list
parse_headers_list(req$headers)
```

send_mail *Send email*

Description

Use the curl SMTP client to send an email. The message argument must be properly formatted RFC2822 email message with From/To/Subject headers and CRLF line breaks.

Usage

```
send_mail(mail_from, mail_rcpt, message, smtp_server = "localhost",
  use_ssl = FALSE, verbose = TRUE, ...)
```

Arguments

mail_from	email address of the sender.
mail_rcpt	one or more recipient email addresses. Do not include names, these go into the message headers.
message	either a string or connection with (properly formatted) email message, including sender/recipient/subject headers. See example.
smtp_server	address of the SMTP server without the smtp:// part
use_ssl	connect with the smtp server over TLS. Gmail requires this.
verbose	print output
...	other options passed to handle_setopt. In most cases you will need to set a username and password to authenticate with the SMTP server.

Examples

```
# Message in RFC2822 format
message <- 'From: "Testbot" <jeroen@ocpu.io>
To: "Jeroen Ooms" <jeroenooms@gmail.com>
Subject: Hello there!

Hi Jeroen,
I am sending this email using curl.'


# Actually send the email
recipients <- c('test@opencpu.org', 'jeroenooms@gmail.com')
sender <- 'test@ocpu.io'
username <- 'mail@ocpu.io'
password <- askpass::askpass(paste("SMTP server password for", username))
send_mail(sender, recipients, smtp_server = 'smtp.mailgun.org',
  message = message, username = username, password = password)
```

Index