

# Package ‘curl’

March 28, 2018

**Type** Package

**Title** A Modern and Flexible Web Client for R

**Version** 3.2

**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://github.com/jeroen/curl#readme> (devel)  
<https://curl.haxx.se/libcurl/> (upstream)

**BugReports** <https://github.com/jeroen/curl/issues>

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

**VignetteBuilder** knitr

**Depends** R (>= 3.0.0)

**LazyData** true

**RoxygenNote** 6.0.1.9000

**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** 2018-03-28 07:42:52 UTC

## R topics documented:

curl . . . . .	2
curl_download . . . . .	4
curl_echo . . . . .	5
curl_escape . . . . .	5
curl_fetch_memory . . . . .	6
curl_options . . . . .	7
handle . . . . .	8
handle_cookies . . . . .	10
ie_proxy . . . . .	11
multi . . . . .	11
multipart . . . . .	13
nslookup . . . . .	14
parse_date . . . . .	14
parse_headers . . . . .	15
<b>Index</b>	<b>16</b>

---

curl	<i>Curl connection interface</i>
------	----------------------------------

---

### 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

<code>url</code>	character string. See examples.
<code>open</code>	character string. How to open the connection if it should be opened initially. Currently only "r" and "rb" are supported.
<code>handle</code>	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	<i>Download file to disk</i>
---------------	------------------------------

---

## 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

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

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

## End(Not run)
```

---

curl_echo	<i>Echo Service</i>
-----------	---------------------

---

**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 <- handle_setform(new_handle(), foo = "blabla", bar = charToRaw("test"),
myfile = form_file(system.file("DESCRIPTION"), "text/description"))
formdata <- curl_echo(h)

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

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

---

curl_escape	<i>URL encoding</i>
-------------	---------------------

---

**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())
```

**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` callback function or open connection object for receiving data. If NULL the entire response content gets buffered and is returned in the done callback.

## 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)
```

**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_version()
```

```
curl_symbols
```

**Arguments**

`filter` string: only return options with string in name

**Format**

A data frame with columns:

**name** Symbol name

**introduced,deprecated,removed** Versions of libcurl

**value** Integer value of symbol

**type** If an option, the type of value it needs

**Examples**

```
# Available options
curl_options()

# List proxy options
curl_options("proxy")

# Sybol table
head(curl_symbols)
# Curl/ssl version info
curl_version()
```

---

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_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 <a href="#">form_file</a> . To list all allowed options, see <a href="#">curl_options</a>
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 <code>do.call()</code> .

**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	<i>Extract cookies from a handle</i>
----------------	--------------------------------------

---

### 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	<i>Internet Explorer proxy settings</i>
----------	---

---

**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	<i>Async Multi Download</i>
-------	-----------------------------

---

**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 <a href="#">handle</a> with preconfigured url option.
done	callback function for completed request. Single argument with response data in same structure as <a href="#">curl_fetch_memory</a> .
fail	callback function called on failed request. Argument contains error message.
data	callback function or open connection object for receiving data. If NULL the entire response content gets buffered and is returned in the done callback.
pool	a multi handle created by <a href="#">new_pool</a> . 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", open = "wb")
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
close(con)
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**

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

---

nslookup	<i>Lookup a hostname</i>
----------	--------------------------

---

### Description

The nslookup function is similar to ns1 but works on all platforms and can resolve ipv6 addresses if supported by the OS. Default behavior raises an error if lookup fails. The has\_internet function tests the internet connection by resolving a random address.

### 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.

### 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	<i>Parse date/time</i>
------------	------------------------

---

### 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	<i>Parse response headers</i>
---------------	-------------------------------

---

## 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

<code>txt</code>	raw or character vector with the header data
<code>multiple</code>	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)
```

# Index

## \*Topic **datasets**

- [curl\\_options](#), 7
- [curl](#), 2, 6, 8
- [curl\\_download](#), 4, 6, 8
- [curl\\_echo](#), 5
- [curl\\_escape](#), 5
- [curl\\_fetch\\_disk \(curl\\_fetch\\_memory\)](#), 6
- [curl\\_fetch\\_memory](#), 6, 6, 8, 12
- [curl\\_fetch\\_multi \(curl\\_fetch\\_memory\)](#), 6
- [curl\\_fetch\\_stream \(curl\\_fetch\\_memory\)](#), 6
- [curl\\_options](#), 7, 9
- [curl\\_symbols \(curl\\_options\)](#), 7
- [curl\\_unescape \(curl\\_escape\)](#), 5
- [curl\\_version \(curl\\_options\)](#), 7
  
- [download.file](#), 4
  
- [form\\_data \(multipart\)](#), 13
- [form\\_file](#), 9
- [form\\_file \(multipart\)](#), 13
  
- [handle](#), 2, 4, 8, 10, 12
- [handle\\_cookies](#), 9, 10
- [handle\\_data \(handle\)](#), 8
- [handle\\_reset \(handle\)](#), 8
- [handle\\_setform \(handle\)](#), 8
- [handle\\_setheaders \(handle\)](#), 8
- [handle\\_setopt \(handle\)](#), 8
- [has\\_internet \(nslookup\)](#), 14
  
- [ie\\_get\\_proxy\\_for\\_url \(ie\\_proxy\)](#), 11
- [ie\\_proxy](#), 11
- [ie\\_proxy\\_info \(ie\\_proxy\)](#), 11
- [isIncomplete](#), 3
  
- [multi](#), 11
- [multi\\_add](#), 12
- [multi\\_add \(multi\)](#), 11
- [multi\\_cancel](#), 12
- [multi\\_cancel \(multi\)](#), 11
  
- [multi\\_fdset](#), 12
- [multi\\_fdset \(multi\)](#), 11
- [multi\\_list \(multi\)](#), 11
- [multi\\_run](#), 12
- [multi\\_run \(multi\)](#), 11
- [multi\\_set \(multi\)](#), 11
- [multipart](#), 13
  
- [new\\_handle \(handle\)](#), 8
- [new\\_pool](#), 6, 12
- [new\\_pool \(multi\)](#), 11
- [nslookup](#), 14
  
- [parse\\_date](#), 14
- [parse\\_headers](#), 15
- [parse\\_headers\\_list \(parse\\_headers\)](#), 15
  
- [strptime](#), 14
  
- [url](#), 2