Package 'httr'

August 29, 2016

Version 1.2.1

Title Tools for Working with URLs and HTTP

Description Useful tools for working with HTTP organised by HTTP verbs (GET(), POST(), etc). Configuration functions make it easy to control additional request components (authenticate(), add_headers() and so on).

Depends R (>= 3.0.0)

Imports jsonlite, mime, curl (>= 0.9.1), openssl (>= 0.8), R6

Suggests httpuv, jpeg, knitr, png, testthat (>= 0.8.0), readr, xml2, rmarkdown

VignetteBuilder knitr

License MIT + file LICENSE

URL https://github.com/hadley/httr

RoxygenNote 5.0.1

NeedsCompilation no

Author Hadley Wickham [aut, cre], RStudio [cph]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2016-07-03 22:33:34

R topics documented:

add_headers *Add additional headers to a request.*

Description

Wikipedia provides a useful list of common http headers: http://en.wikipedia.org/wiki/List_of_HTTP_header_fields.

Usage

```
add_headers(..., .headers = character())
```

Arguments

...	named header values. To stop an existing header from being set, pass an empty string: "".
.headers	a named character vector

See Also

accept and content_type for convenience functions for setting accept and content-type headers.

Other config: authenticate, config, set_cookies, timeout, use_proxy, user_agent, verbose

Examples

```
add_headers(a = 1, b = 2)
add_headers(.headers = c(a = "1", b = "2"))

GET("http://httpbin.org/headers")

# Add arbitrary headers
GET("http://httpbin.org/headers",
 add_headers(version = version$version.string))

# Override default headers with empty strings
GET("http://httpbin.org/headers", add_headers(Accept = ""))
```

authenticate *Use http authentication.*

Description

It's not obvious how to turn authentication off after using it, so I recommend using custom handles with authentication.

Usage

```
authenticate(user, password, type = "basic")
```

Arguments

user	user name
password	password
type	type of HTTP authentication. Should be one of the following types supported by Curl: basic, digest, digest_ie, gssnegotiate, ntlm, any. It defaults to "basic", the most common type.

See Also

Other config: add_headers, config, set_cookies, timeout, use_proxy, user_agent, verbose

Examples

```
GET("http://httpbin.org/basic-auth/user/passwd")
GET("http://httpbin.org/basic-auth/user/passwd",
  authenticate("user", "passwd"))
```

BROWSE *Open specified url in browser.*

Description

(This isn't really a http verb, but it seems to follow the same format).

Usage

```
BROWSE(url = NULL, config = list(), ..., handle = NULL)
```

Arguments

url	the url of the page to retrieve
config	All configuration options are ignored because the request is handled by the browser, not **RCurl**.
...	Further named parameters, such as query, path, etc, passed on to modify_url. Unnamed parameters will be combined with config.
handle	The handle to use with this request. If not supplied, will be retrieved and reused from the handle_pool based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See handle_pool for more details.

Details

Only works in interactive sessions.

See Also

Other http methods: DELETE, GET, HEAD, PATCH, POST, PUT, VERB

Examples

```
BROWSE("http://google.com")
BROWSE("http://had.co.nz")
```

---

cache_info *Compute caching information for a response.*

Description

`cache_info()` gives details of cacheability of a response, `rerequest()` re-performs the original request doing as little work as possible (if not expired, returns response as is, or performs revalidation if Etag or Last-Modified headers are present).

Usage

```
cache_info(r)

rerequest(r)
```

Arguments

r	A response

Examples

```
# Never cached, always causes redownload
r1 <- GET("https://www.google.com")
cache_info(r1)
r1$date
rerequest(r1)$date

# Expires in a year
r2 <- GET("https://www.google.com/images/srpr/logo11w.png")
cache_info(r2)
r2$date
rerequest(r2)$date

# Has last-modified and etag, so does revalidation
r3 <- GET("http://httpbin.org/cache")
cache_info(r3)
r3$date
```

```
rerequest(r3)$date

# Expires after 5 seconds
## Not run:
r4 <- GET("http://httpbin.org/cache/5")
cache_info(r4)
r4$date
rerequest(r4)$date
Sys.sleep(5)
cache_info(r4)
rerequest(r4)$date

## End(Not run)
```

`config` *Set curl options.*

Description

Generally you should only need to use this function to set CURL options directly if there isn't already a helpful wrapper function, like `set_cookies`, `add_headers` or `authenticate`. To use this function effectively requires some knowledge of CURL, and CURL options. Use `httr_options` to see a complete list of available options. To see the libcurl documentation for a given option, use `curl_docs`.

Usage

```
config(..., token = NULL)
```

Arguments

`...`	named Curl options.
`token`	An OAuth token (1.0 or 2.0)

Details

Unlike Curl (and RCurl), all configuration options are per request, not per handle.

See Also

`set_config` to set global config defaults, and `with_config` to temporarily run code with set options.

All known available options are listed in `httr_options`

Other config: `add_headers`, `authenticate`, `set_cookies`, `timeout`, `use_proxy`, `user_agent`, `verbose`

Other ways to set configuration: `set_config`, `with_config`

Examples

```
# There are a number of ways to modify the configuration of a request
# * you can add directly to a request
HEAD("https://www.google.com", verbose())

# * you can wrap with with_config()
with_config(verbose(), HEAD("https://www.google.com"))

# * you can set global with set_config()
old <- set_config(verbose())
HEAD("https://www.google.com")
# and re-establish the previous settings with
set_config(old, override = TRUE)
HEAD("https://www.google.com")
# or
reset_config()
HEAD("https://www.google.com")

# If available, you should use a friendly httr wrapper over RCurl
# options. But you can pass Curl options (as listed in httr_options())
# in config
HEAD("https://www.google.com/", config(verbose = TRUE))
```

---

content *Extract content from a request.*

Description

There are currently three ways to retrieve the contents of a request: as a raw object (`as = "raw"`), as a character vector, (`as = "text"`), and as parsed into an R object where possible, (`as = "parsed"`). If `as` is not specified, `content` does its best to guess which output is most appropriate.

Usage

```
content(x, as = NULL, type = NULL, encoding = NULL, ...)
```

Arguments

`x`	request object
`as`	desired type of output: `raw`, `text` or `parsed`. `content` attempts to automatically figure out which one is most appropriate, based on the content-type.
`type`	MIME type (aka internet media type) used to override the content type returned by the server. See http://en.wikipedia.org/wiki/Internet_media_type for a list of common types.
`encoding`	For text, overrides the charset or the Latin1 (ISO-8859-1) default, if you know that the server is returning the incorrect encoding as the charset in the content-type. Use for text and parsed outputs.
`...`	Other parameters parsed on to the parsing functions, if `as = "parsed"`

Details

content currently knows about the following mime types:

- text/html: read_html
- text/xml: read_xml
- text/csv: read_csv
- text/tab-separated-values: read_tsv
- application/json: fromJSON
- application/x-www-form-urlencoded: parse_query
- image/jpeg: readJPEG
- image/png: readPNG

as = "parsed" is provided as a convenience only: if the type you are trying to parse is not available, use as = "text" and parse yourself.

Value

For "raw", a raw vector.

For "text", a character vector of length 1. The character vector is always re-encoded to UTF-8. If this encoding fails (usually because the page declares an incorrect encoding), content() will return NA.

For "auto", a parsed R object.

WARNING

When using content() in a package, DO NOT use on as = "parsed". Instead, check the mime-type is what you expect, and then parse yourself. This is safer, as you will fail informatively if the API changes, and you will protect yourself against changes to httr.

See Also

Other response methods: http_error, http_status, response, stop_for_status

Examples

```
r <- POST("http://httpbin.org/post", body = list(a = 1, b = 2))
content(r) # automatically parses JSON
cat(content(r, "text"), "\n") # text content
content(r, "raw") # raw bytes from server

rlogo <- content(GET("http://cran.r-project.org/Rlogo.jpg"))
plot(0:1, 0:1, type = "n")
rasterImage(rlogo, 0, 0, 1, 1)
```

content_type *Set content-type and accept headers.*

Description

These are convenient wrappers aroud add_headers.

Usage

```
content_type(type)

content_type_json()

content_type_xml()

accept(type)

accept_json()

accept_xml()
```

Arguments

type	A mime type or a file extension. If a file extension (i.e. starts with .) will guess the mime type using guess_type.

Details

accept_json/accept_xml and content_type_json/content_type_xml are useful shortcuts to ask for json or xml responses or tell the server you are sending json/xml.

Examples

```
GET("http://httpbin.org/headers")

GET("http://httpbin.org/headers", accept_json())
GET("http://httpbin.org/headers", accept("text/csv"))
GET("http://httpbin.org/headers", accept(".doc"))

GET("http://httpbin.org/headers", content_type_xml())
GET("http://httpbin.org/headers", content_type("text/csv"))
GET("http://httpbin.org/headers", content_type(".xml"))
```

cookies *Access cookies in a response.*

Description

Access cookies in a response.

Usage

```
cookies(x)
```

Arguments

x	A response.

See Also

set_cookies() to send cookies in request.

Examples

```
r <- GET("http://httpbin.org/cookies/set", query = list(a = 1, b = 2))
cookies(r)
```

DELETE *Send a DELETE request.*

Description

Send a DELETE request.

Usage

```
DELETE(url = NULL, config = list(), ..., body = NULL,
  encode = c("multipart", "form", "json", "raw"), handle = NULL)
```

Arguments

url	the url of the page to retrieve
config	Additional configuration settings such as http authentication (authenticate), additional headers (add_headers), cookies (set_cookies) etc. See config for full details and list of helpers.
...	Further named parameters, such as query, path, etc, passed on to modify_url. Unnamed parameters will be combined with config.
body	One of the following:

- FALSE: No body. This is typically not used with POST, PUT, or PATCH, but can be useful if you need to send a bodyless request (like GET) with VERB().
- NULL: An empty body
- "": A length 0 body
- upload_file("path/"): The contents of a file. The mime type will be guessed from the extension, or can be supplied explicitly as the second argument to upload_file()
- A character or raw vector: sent as is in body. Use content_type to tell the server what sort of data you are sending.
- A named list: See details for encode.

encode — If the body is a named list, how should it be encoded? Can be one of form (application/x-www-form-urlencoded), multipart, (multipart/form-data), or json (application/json).

For "multipart", list elements can be strings or objects created by upload_file. For "form", elements are coerced to strings and escaped, use I() to prevent double-escaping. For "json", parameters are automatically "unboxed" (i.e. length 1 vectors are converted to scalars). To preserve a length 1 vector as a vector, wrap in I(). For "raw", either a character or raw vector. You'll need to make sure to set the content_type() yourself.

handle — The handle to use with this request. If not supplied, will be retrieved and reused from the handle_pool based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See handle_pool for more details.

RFC2616

The DELETE method requests that the origin server delete the resource identified by the Request-URI. This method MAY be overridden by human intervention (or other means) on the origin server. The client cannot be guaranteed that the operation has been carried out, even if the status code returned from the origin server indicates that the action has been completed successfully. However, the server SHOULD NOT indicate success unless, at the time the response is given, it intends to delete the resource or move it to an inaccessible location.

A successful response SHOULD be 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action has not yet been enacted, or 204 (No Content) if the action has been enacted but the response does not include an entity.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

See Also

Other http methods: BROWSE, GET, HEAD, PATCH, POST, PUT, VERB

Examples

```
DELETE("http://httpbin.org/delete")
POST("http://httpbin.org/delete")
```

GET *GET a url.*

Description

GET a url.

Usage

```
GET(url = NULL, config = list(), ..., handle = NULL)
```

Arguments

url	the url of the page to retrieve
config	Additional configuration settings such as http authentication (authenticate), additional headers (add_headers), cookies (set_cookies) etc. See config for full details and list of helpers.
...	Further named parameters, such as query, path, etc, passed on to modify_url. Unnamed parameters will be combined with config.
handle	The handle to use with this request. If not supplied, will be retrieved and reused from the handle_pool based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See handle_pool for more details.

RFC2616

The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The semantics of the GET method change to a "conditional GET" if the request message includes an If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match, or If-Range header field. A conditional GET method requests that the entity be transferred only under the circumstances described by the conditional header field(s). The conditional GET method is intended to reduce unnecessary network usage by allowing cached entities to be refreshed without requiring multiple requests or transferring data already held by the client.

The semantics of the GET method change to a "partial GET" if the request message includes a Range header field. A partial GET requests that only part of the entity be transferred, as described in http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.35 The partial GET method is intended to reduce unnecessary network usage by allowing partially-retrieved entities to be completed without transferring data already held by the client.

See Also

Other http methods: BROWSE, DELETE, HEAD, PATCH, POST, PUT, VERB

Examples

```
GET("http://google.com/")
GET("http://google.com/", path = "search")
GET("http://google.com/", path = "search", query = list(q = "ham"))

# See what GET is doing with httpbin.org
url <- "http://httpbin.org/get"
GET(url)
GET(url, add_headers(a = 1, b = 2))
GET(url, set_cookies(a = 1, b = 2))
GET(url, add_headers(a = 1, b = 2), set_cookies(a = 1, b = 2))
GET(url, authenticate("username", "password"))
GET(url, verbose())

# You might want to manually specify the handle so you can have multiple
# independent logins to the same website.
google <- handle("http://google.com")
GET(handle = google, path = "/")
GET(handle = google, path = "search")
```

---

handle *Create a handle tied to a particular host.*

Description

This handle preserves settings and cookies across multiple requests. It is the foundation of all requests performed through the httr package, although it will mostly be hidden from the user.

Usage

```
handle(url, cookies = TRUE)
```

Arguments

url	full url to site
cookies	DEPRECATED

Note

Because of the way argument dispatch works in R, using handle() in the http methods (See GET) will cause problems when trying to pass configuration arguments (See examples below). Directly specifying the handle when using http methods is not recommended in general, since the selection of the correct handle is taken care of when the user passes an url (See handle_pool).

Examples

```
handle("http://google.com")
handle("https://google.com")

h <- handle("http://google.com")
GET(handle = h)
# Should see cookies sent back to server
GET(handle = h, config = verbose())

h <- handle("http://google.com", cookies = FALSE)
GET(handle = h)$cookies

## Not run:
# Using the preferred way of configuring the http methods
# will not work when using handle():
GET(handle = h, timeout(10))
# Passing named arguments will work properly:
GET(handle = h, config = list(timeout(10), add_headers(Accept = "")))

## End(Not run)
```

HEAD *Get url HEADers.*

Description

Get url HEADers.

Usage

```
HEAD(url = NULL, config = list(), ..., handle = NULL)
```

Arguments

`url`	the url of the page to retrieve
`config`	Additional configuration settings such as http authentication (`authenticate`), additional headers (`add_headers`), cookies (`set_cookies`) etc. See `config` for full details and list of helpers.
`...`	Further named parameters, such as `query`, `path`, etc, passed on to `modify_url`. Unnamed parameters will be combined with `config`.
`handle`	The handle to use with this request. If not supplied, will be retrieved and reused from the `handle_pool` based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See `handle_pool` for more details.

RFC2616

The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

The response to a HEAD request MAY be cacheable in the sense that the information contained in the response MAY be used to update a previously cached entity from that resource. If the new field values indicate that the cached entity differs from the current entity (as would be indicated by a change in Content-Length, Content-MD5, ETag or Last-Modified), then the cache MUST treat the cache entry as stale.

See Also

Other http methods: BROWSE, DELETE, GET, PATCH, POST, PUT, VERB

Examples

```
HEAD("http://google.com")
headers(HEAD("http://google.com"))
```

---

headers *Extract the headers from a response*

Description

Extract the headers from a response

Usage

```
headers(x)
```

Arguments

x	A request object

See Also

add_headers() to send additional headers in a request

Examples

```
r <- GET("http://httpbin.org/get")
headers(r)
```

`http_error` *Check for an http error.*

Description

Check for an http error.

Usage

```
http_error(x, ...)
```

Arguments

`x`	Object to check. Default methods are provided for strings (which perform an `HEAD` request), responses, and integer status codes.
`...`	Other arguments passed on to methods.

Value

`TRUE` if the request fails (status code 400 or above), otherwise `FALSE`.

See Also

Other response methods: `content`, `http_status`, `response`, `stop_for_status`

Examples

```
# You can pass a url:
http_error("http://www.google.com")
http_error("http://httpbin.org/status/404")

# Or a request
r <- GET("http://httpbin.org/status/201")
http_error(r)

# Or an (integer) status code
http_error(200L)
http_error(404L)
```

`http_status` *Give information on the status of a request.*

Description

Extract the http status code and convert it into a human readable message.

Usage

```
http_status(x)
```

Arguments

`x`	a request object or a number.

Details

http servers send a status code with the response to each request. This code gives information regarding the outcome of the execution of the request on the server. Roughly speaking, codes in the 100s and 200s mean the request was successfully executed; codes in the 300s mean the page was redirected; codes in the 400s mean there was a mistake in the way the client sent the request; codes in the 500s mean the server failed to fulfill an apparently valid request. More details on the codes can be found at `http://en.wikipedia.org/wiki/Http_error_codes`.

Value

If the status code does not match a known status, an error. Otherwise, a list with components

`category`	the broad category of the status
`message`	the meaning of the status code

See Also

Other response methods: `content`, `http_error`, `response`, `stop_for_status`

Examples

```
http_status(100)
http_status(404)

x <- GET("http://httpbin.org/status/200")
http_status(x)

http_status(GET("http://httpbin.org/status/300"))
http_status(GET("http://httpbin.org/status/301"))
http_status(GET("http://httpbin.org/status/404"))

# errors out on unknown status
## Not run: http_status(GET("http://httpbin.org/status/320"))
```

http_type *Extract the content type of a response*

Description

Extract the content type of a response

Usage

```
http_type(x)
```

Arguments

x	A response

Value

A string giving the complete mime type, with all parameters stripped off.

Examples

```
r1 <- GET("http://httpbin.org/image/png")
http_type(r1)
headers(r1)[["Content-Type"]]

r2 <- GET("http://httpbin.org/ip")
http_type(r2)
headers(r2)[["Content-Type"]]
```

httr **httr** *makes http easy.*

Description

`httr` is organised around the five most common http verbs: `GET`, `PATCH`, `POST`, `HEAD`, `PUT`, and `DELETE`.

Details

Each request returns a `response` object which provides easy access to status code, cookies, headers, timings, and other useful info. The content of the request is available as a raw vector (`content`), character vector (`text_content`), or parsed into an R object (`parsed_content`), currently for html, xml, json, png and jpeg).

Requests can be modified by various config options like `set_cookies`, `add_headers`, `authenticate`, `use_proxy`, `verbose`, and `timeout`

httr supports OAuth 1.0 and 2.0. Use `oauth1.0_token` and `oauth2.0_token` to get user tokens, and `sign_oauth1.0` and `sign_oauth2.0` to sign requests. The demos directory has six demos of using OAuth: three for 1.0 (linkedin, twitter and vimeo) and three for 2.0 (facebook, github, google).

`httr_dr` *Diagnose common configuration problems*

Description

Currently one check: that curl uses nss.

Usage

```
httr_dr()
```

`httr_options` *List available options.*

Description

This function lists all available options for `config()`. It provides both the short R name which you use with httr, and the longer Curl name, which is useful when searching the documentation. `curl_doc` opens a link to the libcurl documentation for an option in your browser.

Usage

```
httr_options(matches)

curl_docs(x)
```

Arguments

`matches`	If not missing, this restricts the output so that either the httr or curl option matches this regular expression.
`x`	An option name (either short or full).

Details

RCurl and httr use slightly different names to libcurl: the initial `CURLOPT_` is removed, all underscores are converted to periods and the option is given in lower case. Thus "CURLOPT_SSLENGINE_DEFAULT" becomes "sslengine.default".

Value

A data frame with three columns:

`httr`	The short name used in httr
`libcurl`	The full name used by libcurl
`type`	The type of R object that the option accepts

Examples

```
httr_options()
httr_options("post")

# Use curl_docs to read the curl documentation for each option.
# You can use either the httr or curl option name.
curl_docs("userpwd")
curl_docs("CURLOPT_USERPWD")
```

`modify_url` *Modify a url.*

Description

Modify a url by first parsing it and then replacing components with the non-NULL arguments of this function.

Usage

```
modify_url(url, scheme = NULL, hostname = NULL, port = NULL,
  path = NULL, query = NULL, params = NULL, fragment = NULL,
  username = NULL, password = NULL)
```

Arguments

`url`	the url to modify
`scheme, hostname, port, path, query, params, fragment, username, password`	components of the url to change

`oauth1.0_token` *Generate an oauth1.0 token.*

Description

This is the final object in the OAuth dance - it encapsulates the app, the endpoint, other parameters and the received credentials.

Usage

```
oauth1.0_token(endpoint, app, permission = NULL, as_header = TRUE,
  private_key = NULL, cache = getOption("httr_oauth_cache"))
```

Arguments

`endpoint`	An OAuth endpoint, created by `oauth_endpoint`
`app`	An OAuth consumer application, created by `oauth_app`
`permission`	optional, a string of permissions to ask for.
`as_header`	If `TRUE`, the default, sends oauth in header. If `FALSE`, adds as parameter to url.
`private_key`	Optional, a key provided by `read_key`. Used for signed OAuth 1.0.
`cache`	A logical value or a string. `TRUE` means to cache using the default cache file `.httr-oauth`, `FALSE` means don't cache, and `NA` means to guess using some sensible heuristics. A string mean use the specified path as the cache file.

Details

See `Token` for full details about the token object, and the caching policies used to store credentials across sessions.

Value

A `Token1.0` reference class (RC) object.

See Also

Other OAuth: `oauth2.0_token`, `oauth_app`, `oauth_endpoint`, `oauth_service_token`

`oauth2.0_token` *Generate an oauth2.0 token.*

Description

This is the final object in the OAuth dance - it encapsulates the app, the endpoint, other parameters and the received credentials. It is a reference class so that it can be seamlessly updated (e.g. using `$refresh()`) when access expires.

Usage

```
oauth2.0_token(endpoint, app, scope = NULL, user_params = NULL,
  type = NULL, use_oob = getOption("httr_oob_default"), as_header = TRUE,
  use_basic_auth = FALSE, cache = getOption("httr_oauth_cache"))
```

Arguments

`endpoint`	An OAuth endpoint, created by `oauth_endpoint`
`app`	An OAuth consumer application, created by `oauth_app`
`scope`	a character vector of scopes to request.
`user_params`	Named list holding endpoint specific parameters to pass to the server when posting the request for obtaining or refreshing the access token.
`type`	content type used to override incorrect server response
`use_oob`	if FALSE, use a local webserver for the OAuth dance. Otherwise, provide a URL to the user and prompt for a validation code. Defaults to the of the `"httr_oob_default"` default, or `TRUE` if `httpuv` is not installed.
`as_header`	If `TRUE`, the default, configures the token to add itself to the bearer header of subsequent requests. If `FALSE`, configures the token to add itself as a url parameter of subsequent requests.
`use_basic_auth`	if `TRUE` use http basic authentication to retrieve the token. Some authorization servers require this. If `FALSE`, the default, retrieve the token by including the app key and secret in the request body.
`cache`	A logical value or a string. `TRUE` means to cache using the default cache file `.httr-oauth`, `FALSE` means don't cache, and `NA` means to guess using some sensible heuristics. A string mean use the specified path as the cache file.

Details

See `Token` for full details about the token object, and the caching policies used to store credentials across sessions.

Value

A `Token2.0` reference class (RC) object.

See Also

Other OAuth: oauth1.0_token, oauth_app, oauth_endpoint, oauth_service_token

---

oauth_app *Create an OAuth application.*

Description

The OAuth framework doesn't match perfectly to use from R. Each user of the package for a particular OAuth enabled site must create their own application. See the demos for instructions on how to do this for linkedin, twitter, vimeo, facebook, github and google.

Usage

```
oauth_app(appname, key, secret = NULL)
```

Arguments

appname	name of the application. This is not used for OAuth, but is used to make it easier to identify different applications and provide a consistent way of storing secrets in environment variables.
key	consumer key (equivalent to a user name)
secret	consumer secret. This is not equivalent to a password, and is not really a secret. If you are writing an API wrapper package, it is fine to include this secret in your package code. Use NULL to not store a secret: this is useful if you're relying on cached OAuth tokens.

See Also

Other OAuth: oauth1.0_token, oauth2.0_token, oauth_endpoint, oauth_service_token

Examples

```
## Not run:
# These work on my computer because I have the right envvars set up
linkedin_app <- oauth_app("linkedin", key = "outmkw3859gy")
github_app <- oauth_app("github", "56b637a5baffac62cad9")

## End(Not run)

# If you're relying on caching, supply an explicit NULL to
# suppress the warning message
oauth_app("my_app", "mykey")
oauth_app("my_app", "mykey", NULL)
```

oauth_endpoint *Describe an OAuth endpoint.*

Description

See oauth_endpoints for a list of popular OAuth endpoints baked into httr.

Usage

```
oauth_endpoint(request = NULL, authorize, access, ..., base_url = NULL)
```

Arguments

request	url used to request initial (unauthenticated) token. If using OAuth2.0, leave as NULL.
authorize	url to send client to for authorisation
access	url used to exchange unauthenticated for authenticated token.
...	other additional endpoints.
base_url	option url to use as base for request, authorize and access urls.

See Also

Other OAuth: oauth1.0_token, oauth2.0_token, oauth_app, oauth_service_token

Examples

```
linkedin <- oauth_endpoint("requestToken", "authorize", "accessToken",
  base_url = "https://api.linkedin.com/uas/oauth")
github <- oauth_endpoint(NULL, "authorize", "access_token",
  base_url = "https://github.com/login/oauth")
facebook <- oauth_endpoint(
  authorize = "https://www.facebook.com/dialog/oauth",
  access = "https://graph.facebook.com/oauth/access_token")

oauth_endpoints
```

oauth_endpoints *Popular oauth endpoints.*

Description

Provides some common OAuth endpoints.

Usage

```
oauth_endpoints(name)
```

Arguments

name	One of the following endpoints: linkedin, twitter, vimeo, google, facebook, github, azure.

Examples

```
oauth_endpoints("twitter")
```

---

oauth_service_token *Generate OAuth token for service accounts.*

Description

Service accounts provide a way of using OAuth2 without user intervention. They instead assume that the server has access to a private key used to sign requests. The OAuth app is not needed for service accounts: that information is embedded in the account itself.

Usage

```
oauth_service_token(endpoint, secrets, scope = NULL)
```

Arguments

endpoint	An OAuth endpoint, created by oauth_endpoint
secrets	Secrets loaded from JSON file, downloaded from console.
scope	a character vector of scopes to request.

See Also

Other OAuth: oauth1.0_token, oauth2.0_token, oauth_app, oauth_endpoint

Examples

```
## Not run:
endpoint <- oauth_endpoints("google")
secrets <- jsonlite::fromJSON("~/Desktop/httrtest-45693cbfac92.json")
scope <- "https://www.googleapis.com/auth/bigquery.readonly"

token <- oauth_service_token(endpoint, secrets, scope)

## End(Not run)
```

`parse_http_date` *Parse and print http dates.*

Description

As defined in RFC2616, http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.3, there are three valid formats:

- Sun, 06 Nov 1994 08:49:37 GMT ; RFC 822, updated by RFC 1123
- Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
- Sun Nov 6 08:49:37 1994 ; ANSI C's asctime() format

Usage

```
parse_http_date(x, failure = NA)

http_date(x)
```

Arguments

`x`	For `parse_http_date`, a character vector of strings to parse. All elements must be of the same type. For `http_date`, a `POSIXt` vector.
`failure`	What to return on failure?

Value

A POSIXct object if succesful, otherwise `failure`

Examples

```
parse_http_date("Sun, 06 Nov 1994 08:49:37 GMT")
parse_http_date("Sunday, 06-Nov-94 08:49:37 GMT")
parse_http_date("Sun Nov  6 08:49:37 1994")

http_date(Sys.time())
```

parse_url *Parse and build urls according to RFC1808.*

Description

See http://tools.ietf.org/html/rfc1808.html for details of parsing algorithm.

Usage

```
parse_url(url)

build_url(url)
```

Arguments

url	a character vector (of length 1) to parse into components, or for build_url a url to turn back into a string.

Value

a list containing:

- scheme
- hostname
- port
- path
- params
- fragment
- query, a list
- username
- password

Examples

```
parse_url("http://google.com/")
parse_url("http://google.com:80/")
parse_url("http://google.com:80/?a=1&b=2")

build_url(parse_url("http://google.com/"))
```

PATCH *Send PATCH request to a server.*

Description

Send PATCH request to a server.

Usage

```
PATCH(url = NULL, config = list(), ..., body = NULL,
  encode = c("multipart", "form", "json", "raw"), handle = NULL)
```

Arguments

url	the url of the page to retrieve
config	Additional configuration settings such as http authentication (authenticate), additional headers (add_headers), cookies (set_cookies) etc. See config for full details and list of helpers.
...	Further named parameters, such as query, path, etc, passed on to modify_url. Unnamed parameters will be combined with config.
body	One of the following: • FALSE: No body. This is typically not used with POST, PUT, or PATCH, but can be useful if you need to send a bodyless request (like GET) with VERB(). • NULL: An empty body • "": A length 0 body • upload_file("path/"): The contents of a file. The mime type will be guessed from the extension, or can be supplied explicitly as the second argument to upload_file() • A character or raw vector: sent as is in body. Use content_type to tell the server what sort of data you are sending. • A named list: See details for encode.
encode	If the body is a named list, how should it be encoded? Can be one of form (application/x-www-form-urlencoded), multipart, (multipart/form-data), or json (application/json). For "multipart", list elements can be strings or objects created by upload_file. For "form", elements are coerced to strings and escaped, use I() to prevent double-escaping. For "json", parameters are automatically "unboxed" (i.e. length 1 vectors are converted to scalars). To preserve a length 1 vector as a vector, wrap in I(). For "raw", either a character or raw vector. You'll need to make sure to set the content_type() yourself.
handle	The handle to use with this request. If not supplied, will be retrieved and reused from the handle_pool based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See handle_pool for more details.

See Also

Other http methods: BROWSE, DELETE, GET, HEAD, POST, PUT, VERB

POST *POST file to a server.*

Description

POST file to a server.

Usage

```
POST(url = NULL, config = list(), ..., body = NULL,
  encode = c("multipart", "form", "json", "raw"), handle = NULL)
```

Arguments

url	the url of the page to retrieve
config	Additional configuration settings such as http authentication (authenticate), additional headers (add_headers), cookies (set_cookies) etc. See config for full details and list of helpers.
...	Further named parameters, such as query, path, etc, passed on to modify_url. Unnamed parameters will be combined with config.
body	One of the following: • FALSE: No body. This is typically not used with POST, PUT, or PATCH, but can be useful if you need to send a bodyless request (like GET) with VERB(). • NULL: An empty body • "": A length 0 body • upload_file("path/"): The contents of a file. The mime type will be guessed from the extension, or can be supplied explicitly as the second argument to upload_file() • A character or raw vector: sent as is in body. Use content_type to tell the server what sort of data you are sending. • A named list: See details for encode.
encode	If the body is a named list, how should it be encoded? Can be one of form (application/x-www-form-urlencoded), multipart, (multipart/form-data), or json (application/json). For "multipart", list elements can be strings or objects created by upload_file. For "form", elements are coerced to strings and escaped, use I() to prevent double-escaping. For "json", parameters are automatically "unboxed" (i.e. length 1 vectors are converted to scalars). To preserve a length 1 vector as a vector, wrap in I(). For "raw", either a character or raw vector. You'll need to make sure to set the content_type() yourself.

handle — The handle to use with this request. If not supplied, will be retrieved and reused from the handle_pool based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See handle_pool for more details.

See Also

Other http methods: BROWSE, DELETE, GET, HEAD, PATCH, PUT, VERB

Examples

```
b2 <- "http://httpbin.org/post"
POST(b2, body = "A simple text string")
POST(b2, body = list(x = "A simple text string"))
POST(b2, body = list(y = upload_file(system.file("CITATION"))))
POST(b2, body = list(x = "A simple text string"), encode = "json")

# Various types of empty body:
POST(b2, body = NULL, verbose())
POST(b2, body = FALSE, verbose())
POST(b2, body = "", verbose())
```

---

progress *Add a progress bar.*

Description

Add a progress bar.

Usage

```
progress(type = c("down", "up"), con = stdout())
```

Arguments

type — Type of progress to display: either number of bytes uploaded or downloaded.

con — Connection to send output too. Usually stdout() or stderr.

Examples

```
# If file size is known, you get a progress bar:
x <- GET("http://courses.had.co.nz/12-oscon/slides.zip", progress())
# Otherwise you get the number of bytes downloaded:
x <- GET("http://httpbin.org/drip?numbytes=4000&duration=3", progress())
```

PUT *Send PUT request to server.*

Description

Send PUT request to server.

Usage

```
PUT(url = NULL, config = list(), ..., body = NULL,
  encode = c("multipart", "form", "json", "raw"), handle = NULL)
```

Arguments

`url`	the url of the page to retrieve
`config`	Additional configuration settings such as http authentication (`authenticate`), additional headers (`add_headers`), cookies (`set_cookies`) etc. See `config` for full details and list of helpers.
`...`	Further named parameters, such as `query`, `path`, etc, passed on to `modify_url`. Unnamed parameters will be combined with `config`.
`body`	One of the following: • `FALSE`: No body. This is typically not used with `POST`, `PUT`, or `PATCH`, but can be useful if you need to send a bodyless request (like `GET`) with `VERB()`. • `NULL`: An empty body • `""`: A length 0 body • `upload_file("path/")`: The contents of a file. The mime type will be guessed from the extension, or can be supplied explicitly as the second argument to `upload_file()` • A character or raw vector: sent as is in body. Use `content_type` to tell the server what sort of data you are sending. • A named list: See details for encode.
`encode`	If the body is a named list, how should it be encoded? Can be one of form (application/x-www-form-urlencoded), multipart, (multipart/form-data), or json (application/json). For "multipart", list elements can be strings or objects created by `upload_file`. For "form", elements are coerced to strings and escaped, use `I()` to prevent double-escaping. For "json", parameters are automatically "unboxed" (i.e. length 1 vectors are converted to scalars). To preserve a length 1 vector as a vector, wrap in `I()`. For "raw", either a character or raw vector. You'll need to make sure to set the `content_type()` yourself.
`handle`	The handle to use with this request. If not supplied, will be retrieved and reused from the `handle_pool` based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See `handle_pool` for more details.

See Also

Other http methods: BROWSE, DELETE, GET, HEAD, PATCH, POST, VERB

Examples

```
POST("http://httpbin.org/put")
PUT("http://httpbin.org/put")

b2 <- "http://httpbin.org/put"
PUT(b2, body = "A simple text string")
PUT(b2, body = list(x = "A simple text string"))
PUT(b2, body = list(y = upload_file(system.file("CITATION"))))
PUT(b2, body = list(x = "A simple text string"), encode = "json")
```

---

response *The response object.*

Description

The response object captures all information from a request. It includes fields:

Details

- `url` the url the request was actually sent to (after redirects)
- `handle` the handle associated with the url
- `status_code` the http status code
- `header` a named list of headers returned by the server
- `cookies` a named list of cookies returned by the server
- `content` the body of the response, as raw vector. See `content` for various ways to access the content.
- `time` request timing information
- `config` configuration for the request

See Also

Other response methods: content, http_error, http_status, stop_for_status

RETRY *Retry a request until it succeeds.*

Description

Safely retry a request until it succeeds (returns an HTTP status code below 400). It is designed to be kind to the server: after each failure randomly waits up to twice as long. (Technically it uses exponential backoff with jitter, using the approach outlined in https://www.awsarchitectureblog.com/2015/03/backoff.html.)

Usage

```
RETRY(verb, url = NULL, config = list(), ..., body = NULL,
  encode = c("multipart", "form", "json", "raw"), times = 3,
  pause_base = 1, pause_cap = 60, handle = NULL, quiet = FALSE)
```

Arguments

verb	Name of verb to use.
url	the url of the page to retrieve
config	Additional configuration settings such as http authentication (authenticate), additional headers (add_headers), cookies (set_cookies) etc. See config for full details and list of helpers.
...	Further named parameters, such as query, path, etc, passed on to modify_url. Unnamed parameters will be combined with config.
body	One of the following: • FALSE: No body. This is typically not used with POST, PUT, or PATCH, but can be useful if you need to send a bodyless request (like GET) with VERB(). • NULL: An empty body • "": A length 0 body • upload_file("path/"): The contents of a file. The mime type will be guessed from the extension, or can be supplied explicitly as the second argument to upload_file() • A character or raw vector: sent as is in body. Use content_type to tell the server what sort of data you are sending. • A named list: See details for encode.
encode	If the body is a named list, how should it be encoded? Can be one of form (application/x-www-form-urlencoded), multipart, (multipart/form-data), or json (application/json). For "multipart", list elements can be strings or objects created by upload_file. For "form", elements are coerced to strings and escaped, use I() to prevent double-escaping. For "json", parameters are automatically "unboxed" (i.e. length 1 vectors are converted to scalars). To preserve a length 1 vector as a vector, wrap in I(). For "raw", either a character or raw vector. You'll need to make sure to set the content_type() yourself.

`times` Maximum number of requests to attempt.

`pause_base, pause_cap`
This method uses exponential back-off with full jitter - this means that each request will randomly wait between 0 and `pause_base * 2 ^ attempt` seconds, up to a maximum of `pause_cap` seconds.

`handle` The handle to use with this request. If not supplied, will be retrieved and reused from the `handle_pool` based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See `handle_pool` for more details.

`quiet` If `FALSE`, will print a message displaying how long until the next request.

Value

The last response. Note that if the request doesn't succeed after `times` times this will be a failed request, i.e. you still need to use `stop_for_status()`.

Examples

```
# Succeeds straight away
RETRY("GET", "http://httpbin.org/status/200")
# Never succeeds
RETRY("GET", "http://httpbin.org/status/500")
```

---

revoke_all *Revoke all OAuth tokens in the cache.*

Description

Use this function if you think that your token may have been compromised, e.g. you accidentally uploaded the cache file to github. It's not possible to automatically revoke all tokens - this function will warn when it can't.

Usage

```
revoke_all(cache_path = NA)
```

Arguments

`cache_path` Path to cache file. Defaults to '.httr-oauth' in current directory.

safe_callback *Generate a safe R callback.*

Description

Generate a safe R callback.

Usage

```
safe_callback(f)
```

Arguments

f	A function.

set_config *Set (and reset) global httr configuration.*

Description

Set (and reset) global httr configuration.

Usage

```
set_config(config, override = FALSE)

reset_config()
```

Arguments

config	Settings as generated by add_headers, set_cookies or authenticate.
override	if TRUE, ignore existing settings, if FALSE, combine new config with old.

Value

invisibility, the old global config.

See Also

Other ways to set configuration: config, with_config

Examples

```
GET("http://google.com")
set_config(verbose())
GET("http://google.com")
reset_config()
GET("http://google.com")
```

set_cookies *Set cookies.*

Description

Set cookies.

Usage

```
set_cookies(..., .cookies = character(0))
```

Arguments

...	a named cookie values
.cookies	a named character vector

See Also

cookies() to see cookies in response.

Other config: add_headers, authenticate, config, timeout, use_proxy, user_agent, verbose

Examples

```
set_cookies(a = 1, b = 2)
set_cookies(.cookies = c(a = "1", b = "2"))

GET("http://httpbin.org/cookies")
GET("http://httpbin.org/cookies", set_cookies(a = 1, b = 2))
```

status_code *Extract status code from response.*

Description

Extract status code from response.

Usage

```
status_code(x)
```

Arguments

x	A response

stop_for_status *Take action on http error.*

Description

Converts http errors to R errors or warnings - these should always be used whenever you're creating requests inside a function, so that the user knows why a request has failed.

Usage

```
stop_for_status(x, task = NULL)

warn_for_status(x, task = NULL)

message_for_status(x, task = NULL)
```

Arguments

x	a response, or numeric http code (or other object with status_code method)
task	The text of the message: either NULL or a character vector. If non-NULL, the error message will finish with "Failed to task".

Value

If request was successful, the response (invisibly). Otherwise, raised a classed http error or warning, as generated by http_condition

See Also

http_status and http://en.wikipedia.org/wiki/Http_status_codes for more information on http status codes.

Other response methods: content, http_error, http_status, response

Examples

```
x <- GET("http://httpbin.org/status/200")
stop_for_status(x) # nothing happens
warn_for_status(x)
message_for_status(x)

x <- GET("http://httpbin.org/status/300")
## Not run: stop_for_status(x)
warn_for_status(x)
message_for_status(x)

x <- GET("http://httpbin.org/status/404")
## Not run: stop_for_status(x)
warn_for_status(x)
```

```
message_for_status(x)

# You can provide more information with the task argumgnet
warn_for_status(x, "download spreadsheet")
message_for_status(x, "download spreadsheet")
```

`timeout` *Set maximum request time.*

Description

Set maximum request time.

Usage

```
timeout(seconds)
```

Arguments

`seconds`	number of seconds to wait for a response until giving up. Can not be less than 1 ms.

See Also

Other config: `add_headers`, `authenticate`, `config`, `set_cookies`, `use_proxy`, `user_agent`, `verbose`

Examples

```
## Not run:
GET("http://httpbin.org/delay/3", timeout(1))
GET("http://httpbin.org/delay/1", timeout(2))

## End(Not run)
```

`upload_file` *Upload a file with* `POST` *or* `PUT`.

Description

Upload a file with `POST` or `PUT`.

Usage

```
upload_file(path, type = NULL)
```

Arguments

path	path to file
type	mime type of path. If not supplied, will be guess by guess_type when needed.

Examples

```
citation <- upload_file(system.file("CITATION"))
POST("http://httpbin.org/post", body = citation)
POST("http://httpbin.org/post", body = list(y = citation))
```

---

user_agent *Set user agent.*

Description

Override the default RCurl user agent of NULL

Usage

```
user_agent(agent)
```

Arguments

agent	string giving user agent

See Also

Other config: add_headers, authenticate, config, set_cookies, timeout, use_proxy, verbose

Examples

```
GET("http://httpbin.org/user-agent")
GET("http://httpbin.org/user-agent", user_agent("httr"))
```

---

use_proxy *Use a proxy to connect to the internet.*

Description

Use a proxy to connect to the internet.

Usage

```
use_proxy(url, port = NULL, username = NULL, password = NULL,
  auth = "basic")
```

Arguments

url, port	location of proxy
username, password	login details for proxy, if needed
auth	type of HTTP authentication to use. Should be one of the following: basic, digest, digest_ie, gssnegotiate, ntlm, any.

See Also

Other config: add_headers, authenticate, config, set_cookies, timeout, user_agent, verbose

Examples

```
# See http://www.hidemyass.com/proxy-list for a list of public proxies
# to test with
# GET("http://had.co.nz", use_proxy("64.251.21.73", 8080), verbose())
```

VERB *VERB a url.*

Description

Use an arbitrary verb.

Usage

```
VERB(verb, url = NULL, config = list(), ..., body = NULL,
  encode = c("multipart", "form", "json", "raw"), handle = NULL)
```

Arguments

verb	Name of verb to use.
url	the url of the page to retrieve
config	Additional configuration settings such as http authentication (authenticate), additional headers (add_headers), cookies (set_cookies) etc. See config for full details and list of helpers.
...	Further named parameters, such as query, path, etc, passed on to modify_url. Unnamed parameters will be combined with config.
body	One of the following: • FALSE: No body. This is typically not used with POST, PUT, or PATCH, but can be useful if you need to send a bodyless request (like GET) with VERB(). • NULL: An empty body • "": A length 0 body

- `upload_file("path/")`: The contents of a file. The mime type will be guessed from the extension, or can be supplied explicitly as the second argument to `upload_file()`
- A character or raw vector: sent as is in body. Use `content_type` to tell the server what sort of data you are sending.
- A named list: See details for encode.

`encode` If the body is a named list, how should it be encoded? Can be one of form (application/x-www-form-urlencoded), multipart, (multipart/form-data), or json (application/json).

For "multipart", list elements can be strings or objects created by `upload_file`. For "form", elements are coerced to strings and escaped, use `I()` to prevent double-escaping. For "json", parameters are automatically "unboxed" (i.e. length 1 vectors are converted to scalars). To preserve a length 1 vector as a vector, wrap in `I()`. For "raw", either a character or raw vector. You'll need to make sure to set the `content_type()` yourself.

`handle` The handle to use with this request. If not supplied, will be retrieved and reused from the `handle_pool` based on the scheme, hostname and port of the url. By default **httr** requests to the same scheme/host/port combo. This substantially reduces connection time, and ensures that cookies are maintained over multiple requests to the same host. See `handle_pool` for more details.

See Also

Other http methods: `BROWSE`, `DELETE`, `GET`, `HEAD`, `PATCH`, `POST`, `PUT`

Examples

```
r <- VERB("PROPFIND", "http://svn.r-project.org/R/tags/",
  add_headers(depth = 1), verbose())
stop_for_status(r)
content(r)

VERB("POST", url = "http://httpbin.org/post")
VERB("POST", url = "http://httpbin.org/post", body = "foobar")
```

---

verbose *Give verbose output.*

Description

A verbose connection provides much more information about the flow of information between the client and server.

Usage

```
verbose(data_out = TRUE, data_in = FALSE, info = FALSE, ssl = FALSE)
```

Arguments

data_out	Show data sent to the server.
data_in	Show data recieved 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/recieved over SSL connections?

Prefixes

verbose() uses the following prefixes to distinguish between different components of the http messages:

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

See Also

with_verbose() makes it easier to use verbose mode even when the requests are buried inside another function call.

Other config: add_headers, authenticate, config, set_cookies, timeout, use_proxy, user_agent

Examples

```
GET("http://httpbin.org", verbose())
GET("http://httpbin.org", verbose(info = TRUE))

f <- function() {
  GET("http://httpbin.org")
}
with_verbose(f())
with_verbose(f(), info = TRUE)

# verbose() makes it easy to see exactly what POST requests send
POST_verbose <- function(body, ...) {
  POST("https://httpbin.org/post", body = body, verbose(), ...)
  invisible()
}
POST_verbose(list(x = "a", y = "b"))
POST_verbose(list(x = "a", y = "b"), encode = "form")
POST_verbose(FALSE)
POST_verbose(NULL)
POST_verbose("")
POST_verbose("xyz")
```

with_config *Execute code with configuration set.*

Description

Execute code with configuration set.

Usage

```
with_config(config = config(), expr, override = FALSE)

with_verbose(expr, ...)
```

Arguments

config	Settings as generated by add_headers, set_cookies or authenticate.
expr	code to execute under specified configuration
override	if TRUE, ignore existing settings, if FALSE, combine new config with old.
...	Other arguments passed on to verbose

See Also

Other ways to set configuration: config, set_config

Examples

```
with_config(verbose(), {
  GET("http://had.co.nz")
  GET("http://google.com")
})

# Or even easier:
with_verbose(GET("http://google.com"))
```

write_disk *Control where the response body is written.*

Description

The default behaviour is to use write_memory(), which caches the response locally in memory. This is useful when talking to APIs as it avoids a round-trip to disk. If you want to save a file that's bigger than memory, use write_disk() to save it to a known path.

Usage

```
write_disk(path, overwrite = FALSE)

write_memory()
```

Arguments

path	Path to content to.
overwrite	Will only overwrite existing path if TRUE.

Examples

```
tmp <- tempfile()
r1 <- GET("https://www.google.com", write_disk(tmp))
readLines(tmp)

# The default
r2 <- GET("https://www.google.com", write_memory())

# Save a very large file
## Not run:
GET("http://www2.census.gov/acs2011_5yr/pums/csv_pus.zip",
  write_disk("csv_pus.zip"), progress())

## End(Not run)
```

---

write_stream *Process output in a streaming manner.*

Description

This is the most general way of processing the response from the server - you receive the raw bytes as they come in, and you can do whatever you want with them.

Usage

```
write_stream(f)
```

Arguments

f	Callback function. It should have a single argument, a raw vector containing the bytes recieved from the server. This will usually be 16k or less. The return value of the function is ignored.

Examples

```
GET("https://jeroenooms.github.io/data/diamonds.json",
  write_stream(function(x) {
   print(length(x))
   length(x)
 })
)
```

Index