Package 'httr2'

July 16, 2024

Title Perform HTTP Requests and Process the Responses

Version 1.0.2

Description Tools for creating and modifying HTTP requests, then performing them and processing the results. 'httr2' is a modern re-imagining of 'httr' that uses a pipe-based interface and solves more of the problems that API wrapping packages face.

License MIT + file LICENSE

URL https://httr2.r-lib.org, https://github.com/r-lib/httr2

BugReports https://github.com/r-lib/httr2/issues

Depends R (>= 4.0)

Imports cli (>= 3.0.0), curl (>= 5.1.0), glue, lifecycle, magrittr, openssl, R6, rappdirs, rlang (>= 1.1.0), vctrs (>= 0.6.3), withr

Suggests askpass, bench, clipr, covr, docopt, httpuv, jose, jsonlite, knitr, rmarkdown, testthat (>= 3.1.8), tibble, webfakes, xml2

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Hadley Wickham [aut, cre], Posit Software, PBC [cph, fnd], Maximilian Girlich [ctb]

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2024-07-16 09:30:02 UTC

Contents

`curl_translate` *Translate curl syntax to httr2*

Description

The curl command line tool is commonly used to demonstrate HTTP APIs and can easily be generated from browser developer tools. `curl_translate()` saves you the pain of manually translating these calls by implementing a partial, but frequently used, subset of curl options. Use `curl_help()` to see the supported options, and `curl_translate()` to translate a curl invocation copy and pasted from elsewhere.

Inspired by curlconverter written by Bob Rudis.

Usage

```
curl_translate(cmd, simplify_headers = TRUE)

curl_help()
```

Arguments

`cmd`	Call to curl. If omitted and the clipr package is installed, will be retrieved from the clipboard.
`simplify_headers`	Remove typically unimportant headers included when copying a curl command from the browser. This includes: • `sec-fetch-*` • `sec-ch-ua*` • `referer`, `pragma`, `connection`

Value

A string containing the translated httr2 code. If the input was copied from the clipboard, the translation will be copied back to the clipboard.

Examples

```
curl_translate("curl http://example.com")
curl_translate("curl http://example.com -X DELETE")
curl_translate("curl http://example.com --header A:1 --header B:2")
curl_translate("curl http://example.com --verbose")
```

`iterate_with_offset` *Iteration helpers*

Description

These functions are intended for use with the `next_req` argument to `req_perform_iterative()`. Each implements iteration for a common pagination pattern:

- `iterate_with_offset()` increments a query parameter, e.g. `?page=1`, `?page=2`, or `?offset=1`, `offset=21`.
- `iterate_with_cursor()` updates a query parameter with the value of a cursor found somewhere in the response.
- `iterate_with_link_url()` follows the url found in the `Link` header. See `resp_link_url()` for more details.

Usage

```
iterate_with_offset(
  param_name,
  start = 1,
  offset = 1,
  resp_pages = NULL,
  resp_complete = NULL
)

iterate_with_cursor(param_name, resp_param_value)

iterate_with_link_url(rel = "next")
```

Arguments

`param_name`	Name of query parameter.
`start`	Starting value.
`offset`	Offset for each page. The default is set to `1` so you get (e.g.) `?page=1`, `?page=2`, ... If `param_name` refers to an element index (rather than a page index) you'll want to set this to a larger number so you get (e.g.) `?items=20`, `?items=40`, ...
`resp_pages`	A callback function that takes a response (`resp`) and returns the total number of pages, or `NULL` if unknown. It will only be called once.
`resp_complete`	A callback function that takes a response (`resp`) and returns `TRUE` if there are no further pages.

resp_param_value
: A callback function that takes a response (resp) and returns the next cursor value. Return NULL if there are no further pages.

rel
: The "link relation type" to use to retrieve the next page.

Examples

```
req <- request(example_url()) |>
  req_url_path("/iris") |>
  req_throttle(10) |>
  req_url_query(limit = 50)

# If you don't know the total number of pages in advance, you can
# provide a `resp_complete()` callback
is_complete <- function(resp) {
  length(resp_body_json(resp)$data) == 0
}
resps <- req_perform_iterative(
  req,
  next_req = iterate_with_offset("page_index", resp_complete = is_complete),
  max_reqs = Inf
)

## Not run:
# Alternatively, if the response returns the total number of pages (or you
# can easily calculate it), you can use the `resp_pages()` callback which
# will generate a better progress bar.

resps <- req_perform_iterative(
  req |> req_url_query(limit = 1),
  next_req = iterate_with_offset(
    "page_index",
    resp_pages = function(resp) resp_body_json(resp)$pages
  ),
  max_reqs = Inf
)

## End(Not run)
```

---

last_response *Retrieve most recent request/response*

Description

These functions retrieve the most recent request made by httr2 and the response it received, to facilitate debugging problems *after* they occur. If the request did not succeed (or no requests have been made) last_response() will be NULL.

Usage

```
last_response()

last_request()
```

Value

An HTTP response/request.

Examples

```
invisible(request("http://httr2.r-lib.org") |> req_perform())
last_request()
last_response()
```

---

oauth_cache_path *httr2 OAuth cache location*

Description

When opted-in to, httr2 caches OAuth tokens in this directory. By default, it uses a OS-standard cache directory, but, if needed, you can override the location by setting the HTTR2_OAUTH_CACHE env var.

Usage

```
oauth_cache_path()
```

---

oauth_client *Create an OAuth client*

Description

An OAuth app is the combination of a client, a set of endpoints (i.e. urls where various requests should be sent), and an authentication mechanism. A client consists of at least a `client_id`, and also often a `client_secret`. You'll get these values when you create the client on the API's website.

Usage

```
oauth_client(
  id,
  token_url,
  secret = NULL,
  key = NULL,
  auth = c("body", "header", "jwt_sig"),
  auth_params = list(),
  name = hash(id)
)
```

Arguments

id	Client identifier.
token_url	Url to retrieve an access token.
secret	Client secret. For most apps, this is technically confidential so in principle you should avoid storing it in source code. However, many APIs require it in order to provide a user friendly authentication experience, and the risks of including it are usually low. To make things a little safer, I recommend using obfuscate() when recording the client secret in public code.
key	Client key. As an alternative to using a secret, you can instead supply a confidential private key. This should never be included in a package.
auth	Authentication mechanism used by the client to prove itself to the API. Can be one of three built-in methods ("body", "header", or "jwt"), or a function that will be called with arguments req, client, and the contents of auth_params. The most common mechanism in the wild is "body" where the client_id and (optionally) client_secret are added to the body. "header" sends the client_id and client_secret in HTTP Authorization header. "jwt_sig" will generate a JWT, and include it in a client_assertion field in the body. See oauth_client_req_auth() for more details.
auth_params	Additional parameters passed to the function specified by auth.
name	Optional name for the client. Used when generating the cache directory. If NULL, generated from hash of client_id. If you're defining a client for use in a package, I recommend that you use the package name.

Value

An OAuth client: An S3 list with class httr2_oauth_client.

Examples

```
oauth_client("myclient", "http://example.com/token_url", secret = "DONTLOOK")
```

`oauth_client_req_auth` *OAuth client authentication*

Description

`oauth_client_req_auth()` authenticates a request using the authentication strategy defined by the `auth` and `auth_param` arguments to `oauth_client()`. This is used to authenticate the client as part of the OAuth flow, **not** to authenticate a request on behalf of a user.

There are three built-in strategies:

- `oauth_client_req_body()` adds the client id and (optionally) the secret to the request body, as described in Section 2.3.1 of RFC 6749.
- `oauth_client_req_header()` adds the client id and secret using HTTP basic authentication with the `Authorization` header, as described in Section 2.3.1 of RFC 6749.
- `oauth_client_jwt_rs256()` adds a client assertion to the body using a JWT signed with `jwt_sign_rs256()` using a private key, as described in Section 2.2 of RFC 7523.

You will generally not call these functions directly but will instead specify them through the `auth` argument to `oauth_client()`. The `req` and `client` parameters are automatically filled in; other parameters come from the `auth_params` argument.

Usage

```
oauth_client_req_auth(req, client)

oauth_client_req_auth_header(req, client)

oauth_client_req_auth_body(req, client)

oauth_client_req_auth_jwt_sig(req, client, claim, size = 256, header = list())
```

Arguments

`req`	A request.
`client`	An oauth_client.
`claim`	Claim set produced by `jwt_claim()`.
`size`	Size, in bits, of sha2 signature, i.e. 256, 384 or 512. Only for HMAC/RSA, not applicable for ECDSA keys.
`header`	A named list giving additional fields to include in the JWT header.

Value

A modified HTTP request.

Examples

```
# Show what the various forms of client authentication look like
req <- request("https://example.com/whoami")

client1 <- oauth_client(
  id = "12345",
  secret = "56789",
  token_url = "https://example.com/oauth/access_token",
  name = "oauth-example",
  auth = "body" # the default
)
# calls oauth_client_req_auth_body()
req_dry_run(oauth_client_req_auth(req, client1))

client2 <- oauth_client(
  id = "12345",
  secret = "56789",
  token_url = "https://example.com/oauth/access_token",
  name = "oauth-example",
  auth = "header"
)
# calls oauth_client_req_auth_header()
req_dry_run(oauth_client_req_auth(req, client2))

client3 <- oauth_client(
  id = "12345",
  key = openssl::rsa_keygen(),
  token_url = "https://example.com/oauth/access_token",
  name = "oauth-example",
  auth = "jwt_sig",
  auth_params = list(claim = jwt_claim())
)
# calls oauth_client_req_auth_header_jwt_sig()
req_dry_run(oauth_client_req_auth(req, client3))
```

---

oauth_redirect_uri *Default redirect url for OAuth*

Description

The default redirect uri used by `req_oauth_auth_code()`. Defaults to `http://localhost` unless the `HTTR2_OAUTH_REDIRECT_URL` envvar is set.

Usage

```
oauth_redirect_uri()
```

`oauth_token` *Create an OAuth token*

Description

Creates a S3 object of class <httr2_token> representing an OAuth token returned from the access token endpoint.

Usage

```
oauth_token(
  access_token,
  token_type = "bearer",
  expires_in = NULL,
  refresh_token = NULL,
  ...,
  .date = Sys.time()
)
```

Arguments

`access_token`	The access token used to authenticate request
`token_type`	Type of token; only `"bearer"` is currently supported.
`expires_in`	Number of seconds until token expires.
`refresh_token`	Optional refresh token; if supplied, this can be used to cheaply get a new access token when this one expires.
`...`	Additional components returned by the endpoint
`.date`	Date the request was made; used to convert the relative `expires_in` to an absolute `expires_at`.

Value

An OAuth token: an S3 list with class `httr2_token`.

See Also

`oauth_token_cached()` to use the token cache with a specified OAuth flow.

Examples

```
oauth_token("abcdef")
oauth_token("abcdef", expires_in = 3600)
oauth_token("abcdef", refresh_token = "ghijkl")
```

obfuscate *Obfuscate mildly secret information*

Description

Use `obfuscate("value")` to generate a call to `obfuscated()`, which will unobfuscate the value at the last possible moment. Obfuscated values only work in limited locations:

- The `secret` argument to `oauth_client()`
- Elements of the `data` argument to `req_body_form()`, `req_body_json()`, and `req_body_multipart()`.

Working together this pair of functions provides a way to obfuscate mildly confidential information, like OAuth client secrets. The secret can not be revealed from your inspecting source code, but a skilled R programmer could figure it out with some effort. The main goal is to protect against scraping; there's no way for an automated tool to grab your obfuscated secrets.

Usage

```
obfuscate(x)

obfuscated(x)
```

Arguments

`x`	A string to `obfuscate`, or mark as `obfuscated`.

Value

`obfuscate()` prints the `obfuscated()` call to include in your code. `obfuscated()` returns an S3 class marking the string as obfuscated so it can be unobfuscated when needed.

Examples

```
obfuscate("good morning")

# Every time you obfuscate you'll get a different value because it
# includes 16 bytes of random data which protects against certain types of
# brute force attack
obfuscate("good morning")
```

request *Create a new HTTP request*

Description

To perform a HTTP request, first create a request object with `request()`, then define its behaviour with `req_` functions, then perform the request and fetch the response with `req_perform()`.

Usage

```
request(base_url)
```

Arguments

`base_url`	Base URL for request.

Value

An HTTP request: an S3 list with class `httr2_request`.

Examples

```
request("http://r-project.org")
```

req_auth_basic *Authenticate request with HTTP basic authentication*

Description

This sets the Authorization header. See details at https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization.

Usage

```
req_auth_basic(req, username, password = NULL)
```

Arguments

`req`	A request.
`username`	User name.
`password`	Password. You avoid entering the password directly when calling this function as it will be captured by `.Rhistory`. Instead, leave it unset and the default behaviour will prompt you for it interactively.

Value

A modified HTTP request.

Examples

```
req <- request("http://example.com") |> req_auth_basic("hadley", "SECRET")
req
req |> req_dry_run()

# httr2 does its best to redact the Authorization header so that you don't
# accidentally reveal confidential data. Use `redact_headers` to reveal it:
print(req, redact_headers = FALSE)
req |> req_dry_run(redact_headers = FALSE)

# We do this because the authorization header is not encrypted and the
# so password can easily be discovered:
rawToChar(jsonlite::base64_dec("aGFkbGV5OlNFQ1JFVA=="))
```

---

req_auth_bearer_token *Authenticate request with bearer token*

Description

A bearer token gives the bearer access to confidential resources (so you should keep them secure like you would with a user name and password). They are usually produced by some large authentication scheme (like the various OAuth 2.0 flows), but you are sometimes given then directly.

Usage

```
req_auth_bearer_token(req, token)
```

Arguments

req	A request.
token	A bearer token

Value

A modified HTTP request.

See Also

See RFC 6750 for more details about bearer token usage with OAuth 2.0.

Examples

```
req <- request("http://example.com") |> req_auth_bearer_token("sdaljsdf093lkfs")
req

# httr2 does its best to redact the Authorization header so that you don't
# accidentally reveal confidential data. Use `redact_headers` to reveal it:
print(req, redact_headers = FALSE)
```

---

req_body *Send data in request body*

Description

- `req_body_file()` sends a local file.
- `req_body_raw()` sends a string or raw vector.
- `req_body_json()` sends JSON encoded data. Named components of this data can later be modified with `req_body_json_modify()`.
- `req_body_form()` sends form encoded data.
- `req_body_multipart()` creates a multi-part body.

Adding a body to a request will automatically switch the method to POST.

Usage

```
req_body_raw(req, body, type = NULL)

req_body_file(req, path, type = NULL)

req_body_json(
  req,
  data,
  auto_unbox = TRUE,
  digits = 22,
  null = "null",
  type = "application/json",
  ...
)

req_body_json_modify(req, ...)

req_body_form(.req, ..., .multi = c("error", "comma", "pipe", "explode"))

req_body_multipart(.req, ...)
```

Arguments

req, .req	A request.
body	A literal string or raw vector to send as body.
type	MIME content type. You shouldn't generally need to specify this as the defaults are usually pretty good, e.g. req_body_file() will guess it from the extension of of path. Will be ignored if you have manually set a Content-Type header.
path	Path to file to upload.
data	Data to include in body.
auto_unbox	Should length-1 vectors be automatically "unboxed" to JSON scalars?
digits	How many digits of precision should numbers use in JSON?
null	Should NULL be translated to JSON's null ("null") or an empty list ("list").
...	<dynamic-dots> Name-data pairs used to send data in the body. • For req_body_form(), the values must be strings (or things easily coerced to strings); • For req_body_multipart() the values must be strings or objects produced by curl::form_file()/curl::form_data(). • For req_body_json_modify(), any simple data made from atomic vectors and lists. req_body_json() uses this argument differently; it takes additional arguments passed on to jsonlite::toJSON().
.multi	Controls what happens when an element of ... is a vector containing multiple values: • "error", the default, throws an error. • "comma", separates values with a ,, e.g. ?x=1,2. • "pipe", separates values with a |, e.g. ?x=1|2. • "explode", turns each element into its own parameter, e.g. ?x=1&x=2. If none of these functions work, you can alternatively supply a function that takes a character vector and returns a string.

Value

A modified HTTP request.

Examples

```
req <- request(example_url()) |>
  req_url_path("/post")

# Most APIs expect small amounts of data in either form or json encoded:
req |>
  req_body_form(x = "A simple text string") |>
  req_dry_run()

req |>
```

```
  req_body_json(list(x = "A simple text string")) |>
  req_dry_run()

# For total control over the body, send a string or raw vector
req |>
  req_body_raw("A simple text string") |>
  req_dry_run()

# There are two main ways that APIs expect entire files
path <- tempfile()
writeLines(letters[1:6], path)

# You can send a single file as the body:
req |>
  req_body_file(path) |>
  req_dry_run()

# You can send multiple files, or a mix of files and data
# with multipart encoding
req |>
  req_body_multipart(a = curl::form_file(path), b = "some data") |>
  req_dry_run()
```

`req_cache` *Automatically cache requests*

Description

Use `req_perform()` to automatically cache HTTP requests. Most API requests are not cacheable, but static files often are.

`req_cache()` caches responses to GET requests that have status code 200 and at least one of the standard caching headers (e.g. `Expires`, `Etag`, `Last-Modified`, `Cache-Control`), unless caching has been expressly prohibited with `Cache-Control: no-store`. Typically, a request will still be sent to the server to check that the cached value is still up-to-date, but it will not need to re-download the body value.

To learn more about HTTP caching, I recommend the MDN article HTTP caching.

Usage

```
req_cache(
  req,
  path,
  use_on_error = FALSE,
  debug = getOption("httr2_cache_debug", FALSE),
  max_age = Inf,
  max_n = Inf,
  max_size = 1024^3
)
```

Arguments

`req`	A request.
`path`	Path to cache directory. httr2 doesn't provide helpers to manage the cache, but if you want to empty it, you can use something like `unlink(dir(cache_path, full.names = TRUE))`.
`use_on_error`	If the request errors, and there's a cache response, should `req_perform()` return that instead of generating an error?
`debug`	When `TRUE` will emit useful messages telling you about cache hits and misses. This can be helpful to understand whether or not caching is actually doing anything for your use case.
`max_n, max_age, max_size`	Automatically prune the cache by specifying one or more of: • `max_age`: to delete files older than this number of seconds. • `max_n`: to delete files (from oldest to newest) to preserve at most this many files. • `max_size`: to delete files (from oldest to newest) to preserve at most this many bytes. The cache pruning is performed at most once per minute.

Value

A modified HTTP request.

Examples

```
# GitHub uses HTTP caching for all raw files.
url <- paste0(
  "https://raw.githubusercontent.com/allisonhorst/palmerpenguins/",
  "master/inst/extdata/penguins.csv"
)
# Here I set debug = TRUE so you can see what's happening
req <- request(url) |> req_cache(tempdir(), debug = TRUE)

# First request downloads the data
resp <- req |> req_perform()

# Second request retrieves it from the cache
resp <- req |> req_perform()
```

`req_cookie_preserve` *Preserve cookies across requests*

Description

By default, httr2 uses a clean slate for every request meaning that cookies are not automatically preserved across requests. To preserve cookies, you must set a cookie file which will be read before and updated after each request.

Usage

```
req_cookie_preserve(req, path)
```

Arguments

req	A request.
path	A path to a file where cookies will be read from before and updated after the request.

Examples

```
path <- tempfile()
httpbin <- request(example_url()) |>
  req_cookie_preserve(path)

# Manually set two cookies
httpbin |>
  req_template("/cookies/set/:name/:value", name = "chocolate", value = "chip") |>
  req_perform() |>
  resp_body_json()

httpbin |>
  req_template("/cookies/set/:name/:value", name = "oatmeal", value = "raisin") |>
  req_perform() |>
  resp_body_json()

# The cookie path has a straightforward format
cat(readChar(path, nchars = 1e4))
```

---

req_dry_run *Perform a dry run*

Description

This shows you exactly what httr2 will send to the server, without actually sending anything. It requires the httpuv package because it works by sending the real HTTP request to a local webserver, thanks to the magic of `curl::curl_echo()`.

Usage

```
req_dry_run(req, quiet = FALSE, redact_headers = TRUE)
```

Arguments

req	A request.
quiet	If TRUE doesn't print anything.
redact_headers	Redact confidential data in the headers? Currently redacts the contents of the Authorization header to prevent you from accidentally leaking credentials when debugging/reprexing.

Value

Invisibly, a list containing information about the request, including `method`, `path`, and `headers`.

Examples

```
# httr2 adds default User-Agent, Accept, and Accept-Encoding headers
request("http://example.com") |> req_dry_run()

# the Authorization header is automatically redacted to avoid leaking
# credentials on the console
req <- request("http://example.com") |> req_auth_basic("user", "password")
req |> req_dry_run()

# if you need to see it, use redact_headers = FALSE
req |> req_dry_run(redact_headers = FALSE)
```

---

req_error *Control handling of HTTP errors*

Description

`req_perform()` will automatically convert HTTP errors (i.e. any 4xx or 5xx status code) into R errors. Use `req_error()` to either override the defaults, or extract additional information from the response that would be useful to expose to the user.

Usage

```
req_error(req, is_error = NULL, body = NULL)
```

Arguments

`req`	A request.
`is_error`	A predicate function that takes a single argument (the response) and returns `TRUE` or `FALSE` indicating whether or not an R error should be signalled.
`body`	A callback function that takes a single argument (the response) and returns a character vector of additional information to include in the body of the error. This vector is passed along to the `message` argument of `rlang::abort()` so you can use any formatting that it supports.

Value

A modified HTTP request.

Error handling

`req_perform()` is designed to succeed if and only if you get a valid HTTP response. There are two ways a request can fail:

- The HTTP request might fail, for example if the connection is dropped or the server doesn't exist. This type of error will have class `c("httr2_failure", "httr2_error")`.
- The HTTP request might succeed, but return an HTTP status code that represents an error, e.g. a `404 Not Found` if the specified resource is not found. This type of error will have (e.g.) class `c("httr2_http_404", "httr2_http", "httr2_error")`.

These error classes are designed to be used in conjunction with R's condition handling tools (https://adv-r.hadley.nz/conditions.html). For example, if you want to return a default value when the server returns a 404, use `tryCatch()`:

```
tryCatch(
  req |> req_perform() |> resp_body_json(),
  httr2_http_404 = function(cnd) NULL
)
```

Or if you want to re-throw the error with some additional context, use `withCallingHandlers()`, e.g.:

```
withCallingHandlers(
  req |> req_perform() |> resp_body_json(),
  httr2_http_404 = function(cnd) {
    rlang::abort("Couldn't find user", parent = cnd)
  }
)
```

Learn more about error chaining at rlang::topic-error-chaining.

See Also

`req_retry()` to control when errors are automatically retried.

Examples

```
# Performing this request usually generates an error because httr2
# converts HTTP errors into R errors:
req <- request(example_url()) |>
  req_url_path("/status/404")
try(req |> req_perform())
# You can still retrieve it with last_response()
last_response()

# But you might want to suppress this behaviour:
resp <- req |>
  req_error(is_error = \(resp) FALSE) |>
  req_perform()
```

```
resp

# Or perhaps you're working with a server that routinely uses the
# wrong HTTP error codes only 500s are really errors
request("http://example.com") |>
  req_error(is_error = \(resp) resp_status(resp) == 500)

# Most typically you'll use req_error() to add additional information
# extracted from the response body (or sometimes header):
error_body <- function(resp) {
  resp_body_json(resp)$error
}
request("http://example.com") |>
  req_error(body = error_body)
# Learn more in https://httr2.r-lib.org/articles/wrapping-apis.html
```

`req_headers` *Modify request headers*

Description

`req_headers()` allows you to set the value of any header.

Usage

```
req_headers(.req, ..., .redact = NULL)
```

Arguments

`.req`	A request.
`...`	<dynamic-dots> Name-value pairs of headers and their values. • Use `NULL` to reset a value to httr2's default • Use `""` to remove a header • Use a character vector to repeat a header.
`.redact`	Headers to redact. If `NULL`, the default, the added headers are not redacted.

Value

A modified HTTP request.

Examples

```
req <- request("http://example.com")

# Use req_headers() to add arbitrary additional headers to the request
req |>
  req_headers(MyHeader = "MyValue") |>
  req_dry_run()
```

```
# Repeated use overrides the previous value:
req |>
  req_headers(MyHeader = "Old value") |>
  req_headers(MyHeader = "New value") |>
  req_dry_run()

# Setting Accept to NULL uses curl's default:
req |>
  req_headers(Accept = NULL) |>
  req_dry_run()

# Setting it to "" removes it:
req |>
  req_headers(Accept = "") |>
  req_dry_run()

# If you need to repeat a header, provide a vector of values
# (this is rarely needed, but is important in a handful of cases)
req |>
  req_headers(HeaderName = c("Value 1", "Value 2", "Value 3")) |>
  req_dry_run()

# If you have headers in a list, use !!!
headers <- list(HeaderOne = "one", HeaderTwo = "two")
req |>
   req_headers(!!!headers, HeaderThree = "three") |>
   req_dry_run()

# Use `.redact` to hide a header in the output
req |>
 req_headers(Secret = "this-is-private", Public = "but-this-is-not", .redact = "Secret") |>
  req_dry_run()
```

`req_method` *Set HTTP method in request*

Description

Use this function to use a custom HTTP method like `HEAD`, `DELETE`, `PATCH`, `UPDATE`, or `OPTIONS`. The default method is `GET` for requests without a body, and `POST` for requests with a body.

Usage

```
req_method(req, method)
```

Arguments

`req`	A request.
`method`	Custom HTTP method

Value

A modified HTTP request.

Examples

```
request(example_url()) |> req_method("PATCH")
request(example_url()) |> req_method("PUT")
request(example_url()) |> req_method("HEAD")
```

---

req_oauth_auth_code *OAuth with authorization code*

Description

Authenticate using the OAuth **authorization code flow**, as defined by Section 4.1 of RFC 6749.

This flow is the most commonly used OAuth flow where the user opens a page in their browser, approves the access, and then returns to R. When possible, it redirects the browser back to a temporary local webserver to capture the authorization code. When this is not possible (e.g. when running on a hosted platform like RStudio Server), provide a custom `redirect_uri` and httr2 will prompt the user to enter the code manually.

Learn more about the overall OAuth authentication flow in `vignette("oauth")`.

Usage

```
req_oauth_auth_code(
  req,
  client,
  auth_url,
  scope = NULL,
  pkce = TRUE,
  auth_params = list(),
  token_params = list(),
  redirect_uri = oauth_redirect_uri(),
  cache_disk = FALSE,
  cache_key = NULL,
  host_name = deprecated(),
  host_ip = deprecated(),
  port = deprecated()
)

oauth_flow_auth_code(
  client,
  auth_url,
  scope = NULL,
  pkce = TRUE,
  auth_params = list(),
```

```
  token_params = list(),
  redirect_uri = oauth_redirect_uri(),
  host_name = deprecated(),
  host_ip = deprecated(),
  port = deprecated()
)
```

Arguments

req	A request.
client	An oauth_client().
auth_url	Authorization url; you'll need to discover this by reading the documentation.
scope	Scopes to be requested from the resource owner.
pkce	Use "Proof Key for Code Exchange"? This adds an extra layer of security and should always be used if supported by the server.
auth_params	A list containing additional parameters passed to oauth_flow_auth_code_url().
token_params	List containing additional parameters passed to the token_url.
redirect_uri	URL to redirect back to after authorization is complete. Often this must be registered with the API in advance. httr2 supports three forms of redirect. Firstly, you can use a localhost url (the default), where httr2 will set up a temporary webserver to listen for the OAuth redirect. In this case, httr2 will automatically append a random port. If you need to set it to a fixed port because the API requires it, then specify it with (e.g.) "http://localhost:1011". This technique works well when you are working on your own computer. Secondly, you can provide a URL to a website that uses Javascript to give the user a code to copy and paste back into the R session (see https://www.tidyverse.org/google-callback/ and https://github.com/r-lib/gargle/blob/main/inst/pseudo-oob/google-callback/index.html for examples). This is less convenient (because it requires more user interaction) but also works in hosted environments like RStudio Server. Finally, hosted platforms might set the HTTR2_OAUTH_REDIRECT_URL and HTTR2_OAUTH_CODE_SOURCE_UR environment variables. In this case, httr2 will use HTTR2_OAUTH_REDIRECT_URL for redirects by default, and poll the HTTR2_OAUTH_CODE_SOURCE_URL endpoint with the state parameter until it receives a code in the response (or encounters an error). This delegates completion of the authorization flow to the hosted platform.
cache_disk	Should the access token be cached on disk? This reduces the number of times that you need to re-authenticate at the cost of storing access credentials on disk. Learn more in vignette("oauth")
cache_key	If you want to cache multiple tokens per app, use this key to disambiguate them.
host_name, host_ip, port	**[Deprecated]** Now use redirect_uri instead.

Value

`req_oauth_auth_code()` returns a modified HTTP request that will use OAuth; `oauth_flow_auth_code()` returns an oauth_token.

Security considerations

The authorization code flow is used for both web applications and native applications (which are equivalent to R packages). RFC 8252 spells out important considerations for native apps. Most importantly there's no way for native apps to keep secrets from their users. This means that the server should either not require a `client_secret` (i.e. a public client not an confidential client) or ensure that possession of the `client_secret` doesn't bestow any meaningful rights.

Only modern APIs from the bigger players (Azure, Google, etc) explicitly native apps. However, in most cases, even for older APIs, possessing the `client_secret` gives you no ability to do anything harmful, so our general principle is that it's fine to include it in an R package, as long as it's mildly obfuscated to protect it from credential scraping. There's no incentive to steal your client credentials if it takes less time to create a new client than find your client secret.

See Also

`oauth_flow_auth_code_url()` for the components necessary to write your own auth code flow, if the API you are wrapping does not adhere closely to the standard.

Other OAuth flows: `req_oauth_bearer_jwt()`, `req_oauth_client_credentials()`, `req_oauth_password()`, `req_oauth_refresh()`

Examples

```
req_auth_github <- function(req) {
  req_oauth_auth_code(
    req,
    client = example_github_client(),
    auth_url = "https://github.com/login/oauth/authorize"
  )
}

request("https://api.github.com/user") |>
  req_auth_github()
```

---

req_oauth_bearer_jwt *OAuth with a bearer JWT (JSON web token)*

Description

Authenticate using a **Bearer JWT** (JSON web token) as an authorization grant to get an access token, as defined by Section 2.1 of RFC 7523. It is often used for service accounts, accounts that are used primarily in automated environments.

Learn more about the overall OAuth authentication flow in `vignette("oauth")`.

Usage

```
req_oauth_bearer_jwt(
  req,
  client,
  claim,
  signature = "jwt_encode_sig",
  signature_params = list(),
  scope = NULL,
  token_params = list()
)

oauth_flow_bearer_jwt(
  client,
  claim,
  signature = "jwt_encode_sig",
  signature_params = list(),
  scope = NULL,
  token_params = list()
)
```

Arguments

req	A request.
client	An oauth_client().
claim	A list of claims. If all elements of the claim set are static apart from iat, nbf, exp, or jti, provide a list and jwt_claim() will automatically fill in the dynamic components. If other components need to vary, you can instead provide a zero-argument callback function which should call jwt_claim().
signature	Function use to sign claim, e.g. jwt_encode_sig().
signature_params	Additional arguments passed to signature, e.g. size, header.
scope	Scopes to be requested from the resource owner.
token_params	List containing additional parameters passed to the token_url.

Value

req_oauth_bearer_jwt() returns a modified HTTP request that will use OAuth; oauth_flow_bearer_jwt() returns an oauth_token.

See Also

Other OAuth flows: req_oauth_auth_code(), req_oauth_client_credentials(), req_oauth_password(), req_oauth_refresh()

Examples

```
req_auth <- function(req) {
  req_oauth_bearer_jwt(
    req,
    client = oauth_client("example", "https://example.com/get_token"),
    claim = jwt_claim()
  )
}

request("https://example.com") |>
 req_auth()
```

`req_oauth_client_credentials` *OAuth with client credentials*

Description

Authenticate using OAuth **client credentials flow**, as defined by Section 4.4 of RFC 6749. It is used to allow the client to access resources that it controls directly, not on behalf of an user.

Learn more about the overall OAuth authentication flow in `vignette("oauth")`.

Usage

```
req_oauth_client_credentials(req, client, scope = NULL, token_params = list())

oauth_flow_client_credentials(client, scope = NULL, token_params = list())
```

Arguments

`req`	A request.
`client`	An `oauth_client()`.
`scope`	Scopes to be requested from the resource owner.
`token_params`	List containing additional parameters passed to the `token_url`.

Value

`req_oauth_client_credentials()` returns a modified HTTP request that will use OAuth; `oauth_flow_client_credenti` returns an oauth_token.

See Also

Other OAuth flows: `req_oauth_auth_code()`, `req_oauth_bearer_jwt()`, `req_oauth_password()`, `req_oauth_refresh()`

Examples

```
req_auth <- function(req) {
  req_oauth_client_credentials(
    req,
    client = oauth_client("example", "https://example.com/get_token")
  )
}

request("https://example.com") |>
  req_auth()
```

`req_oauth_device` *OAuth with device flow*

Description

Authenticate using the OAuth **device flow**, as defined by RFC 8628. It's designed for devices that don't have access to a web browser (if you've ever authenticated an app on your TV, this is probably the flow you've used), but it also works well from within R.

Learn more about the overall OAuth authentication flow in `vignette("oauth")`.

Usage

```
req_oauth_device(
  req,
  client,
  auth_url,
  scope = NULL,
  auth_params = list(),
  token_params = list(),
  cache_disk = FALSE,
  cache_key = NULL
)

oauth_flow_device(
  client,
  auth_url,
  pkce = FALSE,
  scope = NULL,
  auth_params = list(),
  token_params = list()
)
```

Arguments

`req`	A request.
`client`	An `oauth_client()`.

auth_url	Authorization url; you'll need to discover this by reading the documentation.
scope	Scopes to be requested from the resource owner.
auth_params	A list containing additional parameters passed to oauth_flow_auth_code_url().
token_params	List containing additional parameters passed to the token_url.
cache_disk	Should the access token be cached on disk? This reduces the number of times that you need to re-authenticate at the cost of storing access credentials on disk. Learn more in vignette("oauth")
cache_key	If you want to cache multiple tokens per app, use this key to disambiguate them.
pkce	Use "Proof Key for Code Exchange"? This adds an extra layer of security and should always be used if supported by the server.

Value

req_oauth_device() returns a modified HTTP request that will use OAuth; oauth_flow_device() returns an oauth_token.

Examples

```
req_auth_github <- function(req) {
  req_oauth_device(
    req,
    client = example_github_client(),
    auth_url = "https://github.com/login/device/code"
  )
}

request("https://api.github.com/user") |>
  req_auth_github()
```

---

req_oauth_password *OAuth with username and password*

Description

This function implements the OAuth **resource owner password flow**, as defined by Section 4.3 of RFC 6749. It allows the user to supply their password once, exchanging it for an access token that can be cached locally.

Learn more about the overall OAuth authentication flow in vignette("oauth").

Usage

```
req_oauth_password(
  req,
  client,
  username,
  password = NULL,
```

```
  scope = NULL,
  token_params = list(),
  cache_disk = FALSE,
  cache_key = username
)

oauth_flow_password(
  client,
  username,
  password = NULL,
  scope = NULL,
  token_params = list()
)
```

Arguments

req	A request.
client	An oauth_client().
username	User name.
password	Password. You avoid entering the password directly when calling this function as it will be captured by .Rhistory. Instead, leave it unset and the default behaviour will prompt you for it interactively.
scope	Scopes to be requested from the resource owner.
token_params	List containing additional parameters passed to the token_url.
cache_disk	Should the access token be cached on disk? This reduces the number of times that you need to re-authenticate at the cost of storing access credentials on disk. Learn more in vignette("oauth")
cache_key	If you want to cache multiple tokens per app, use this key to disambiguate them.

Value

req_oauth_password() returns a modified HTTP request that will use OAuth; oauth_flow_password() returns an oauth_token.

See Also

Other OAuth flows: req_oauth_auth_code(), req_oauth_bearer_jwt(), req_oauth_client_credentials(), req_oauth_refresh()

Examples

```
req_auth <- function(req) {
  req_oauth_password(req,
    client = oauth_client("example", "https://example.com/get_token"),
    username = "username"
  )
}
if (interactive()) {
```

```
  request("https://example.com") |>
    req_auth()
}
```

`req_oauth_refresh` *OAuth with a refresh token*

Description

Authenticate using a **refresh token**, following the process described in Section 6 of RFC 6749.

This technique is primarily useful for testing: you can manually retrieve a OAuth token using another OAuth flow (e.g. with `oauth_flow_auth_code()`), extract the refresh token from the result, and then save in an environment variable for use in automated tests.

When requesting an access token, the server may also return a new refresh token. If this happens, `oauth_flow_refresh()` will warn, and you'll have retrieve a new update refresh token and update the stored value. If you find this happening a lot, it's a sign that you should be using a different flow in your automated tests.

Learn more about the overall OAuth authentication flow in `vignette("oauth")`.

Usage

```
req_oauth_refresh(
  req,
  client,
  refresh_token = Sys.getenv("HTTR2_REFRESH_TOKEN"),
  scope = NULL,
  token_params = list()
)

oauth_flow_refresh(
  client,
  refresh_token = Sys.getenv("HTTR2_REFRESH_TOKEN"),
  scope = NULL,
  token_params = list()
)
```

Arguments

`req`	A request.
`client`	An `oauth_client()`.
`refresh_token`	A refresh token. This is equivalent to a password so shouldn't be typed into the console or stored in a script. Instead, we recommend placing in an environment variable; the default behaviour is to look in `HTTR2_REFRESH_TOKEN`.
`scope`	Scopes to be requested from the resource owner.
`token_params`	List containing additional parameters passed to the `token_url`.

Value

`req_oauth_refresh()` returns a modified HTTP request that will use OAuth; `oauth_flow_refresh()` returns an oauth_token.

See Also

Other OAuth flows: `req_oauth_auth_code()`, `req_oauth_bearer_jwt()`, `req_oauth_client_credentials()`, `req_oauth_password()`

Examples

```
client <- oauth_client("example", "https://example.com/get_token")
req <- request("https://example.com")
req |> req_oauth_refresh(client)
```

---

req_options *Set arbitrary curl options in request*

Description

`req_options()` is for expert use only; it allows you to directly set libcurl options to access features that are otherwise not available in httr2.

Usage

```
req_options(.req, ...)
```

Arguments

`.req`	A request.
`...`	<dynamic-dots> Name-value pairs. The name should be a valid curl option, as found in `curl::curl_options()`.

Value

A modified HTTP request.

Examples

```
# req_options() allows you to access curl options that are not otherwise
# exposed by httr2. For example, in very special cases you may need to
# turn off SSL verification. This is generally a bad idea so httr2 doesn't
# provide a convenient wrapper, but if you really know what you're doing
# you can still access this libcurl option:
req <- request("https://example.com") |>
  req_options(ssl_verifypeer = 0)
```

`req_perform` *Perform a request to get a response*

Description

After preparing a request, call `req_perform()` to perform it, fetching the results back to R as a response.

The default HTTP method is `GET` unless a body (set by req_body_json and friends) is present, in which case it will be `POST`. You can override these defaults with `req_method()`.

Usage

```
req_perform(
  req,
  path = NULL,
  verbosity = NULL,
  mock = getOption("httr2_mock", NULL),
  error_call = current_env()
)
```

Arguments

`req`	A request.
`path`	Optionally, path to save body of the response. This is useful for large responses since it avoids storing the response in memory.
`verbosity`	How much information to print? This is a wrapper around `req_verbose()` that uses an integer to control verbosity: • `0`: no output • `1`: show headers • `2`: show headers and bodies • `3`: show headers, bodies, and curl status messages. Use `with_verbosity()` to control the verbosity of requests that you can't affect directly.
`mock`	A mocking function. If supplied, this function is called with the request. It should return either `NULL` (if it doesn't want to handle the request) or a response (if it does). See `with_mock()`/ `local_mock()` for more details.
`error_call`	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.

Value

- If the HTTP request succeeds, and the status code is ok (e.g. 200), an HTTP response.

- If the HTTP request succeeds, but the status code is an error (e.g a 404), an error with class `c("httr2_http_404", "httr2_http")`. By default, all 400 and 500 status codes will be treated as an error, but you can customise this with `req_error()`.
- If the HTTP request fails (e.g. the connection is dropped or the server doesn't exist), an error with class `"httr2_failure"`.

Requests

Note that one call to `req_perform()` may perform multiple HTTP requests:

- If the `url` is redirected with a 301, 302, 303, or 307, curl will automatically follow the `Location` header to the new location.
- If you have configured retries with `req_retry()` and the request fails with a transient problem, `req_perform()` will try again after waiting a bit. See `req_retry()` for details.
- If you are using OAuth, and the cached token has expired, `req_perform()` will get a new token either using the refresh token (if available) or by running the OAuth flow.

Progress bar

`req_perform()` will automatically add a progress bar if it needs to wait between requests for `req_throttle()` or `req_retry()`. You can turn the progress bar off (and just show the total time to wait) by setting `options(httr2_progress = FALSE)`.

See Also

`req_perform_parallel()` to perform multiple requests in parallel. `req_perform_iterative()` to perform multiple requests iteratively.

Examples

```
request("https://google.com") |>
  req_perform()
```

---

req_perform_iterative *Perform requests iteratively, generating new requests from previous responses*

Description

[Experimental]

`req_perform_iterative()` iteratively generates and performs requests, using a callback function, `next_req`, to define the next request based on the current request and response. You will probably want to pair it with an iteration helper and use a multi-response handler to process the result.

Usage

```
req_perform_iterative(
  req,
  next_req,
  path = NULL,
  max_reqs = 20,
  on_error = c("stop", "return"),
  progress = TRUE
)
```

Arguments

req	The first request to perform.
next_req	A function that takes the previous response (resp) and request (req) and returns a request for the next page or NULL if the iteration should terminate. See below for more details.
path	Optionally, path to save the body of request. This should be a glue string that uses {i} to distinguish different requests. Useful for large responses because it avoids storing the response in memory.
max_reqs	The maximum number of requests to perform. Use Inf to perform all requests until next_req() returns NULL.
on_error	What should happen if a request fails? • "stop", the default: stop iterating with an error. • "return": stop iterating, returning all the successful responses so far, as well as an error object for the failed request.
progress	Display a progress bar? Use TRUE to turn on a basic progress bar, use a string to give it a name, or see progress_bars to customise it in other ways.

Value

A list, at most length max_reqs, containing responses and possibly one error object, if on_error is "return" and one of the requests errors. If present, the error object will always be the last element in the list.

Only httr2 errors are captured; see req_error() for more details.

next_req()

The key piece that makes req_perform_iterative() work is the next_req() argument. For most common cases, you can use one of the canned helpers, like iterate_with_offset(). If, however, the API you're wrapping uses a different pagination system, you'll need to write your own. This section gives some advice.

Generally, your function needs to inspect the response, extract some data from it, then use that to modify the previous request. For example, imagine that the response returns a cursor, which needs to be added to the body of the request. The simplest version of this function might look like this:

```
next_req <- function(resp, req) {
  cursor <- resp_body_json(resp)$next_cursor
  req |> req_body_json_modify(cursor = cursor)
}
```

There's one problem here: if there are no more pages to return, then `cursor` will be `NULL`, but `req_body_json_modify()` will still generate a meaningful request. So we need to handle this specifically by returning `NULL`:

```
next_req <- function(resp, req) {
  cursor <- resp_body_json(resp)$next_cursor
  if (is.null(cursor))
    return(NULL)
  req |> req_body_json_modify(cursor = cursor)
}
```

A value of `NULL` lets `req_perform_iterative()` know there are no more pages remaining.

There's one last feature you might want to add to your iterator: if you know the total number of pages, then it's nice to let `req_perform_iterative()` know so it can adjust the progress bar. (This will only ever decrease the number of pages, not increase it.) You can signal the total number of pages by calling `signal_total_pages()`, like this:

```
next_req <- function(resp, req) {
  body <- resp_body_json(resp)
  cursor <- body$next_cursor
  if (is.null(cursor))
    return(NULL)

  signal_total_pages(body$pages)
  req |> req_body_json_modify(cursor = cursor)
}
```

Examples

```
req <- request(example_url()) |>
  req_url_path("/iris") |>
  req_throttle(10) |>
  req_url_query(limit = 5)

resps <- req_perform_iterative(req, iterate_with_offset("page_index"))

resps |> resps_data(function(resp) {
  data <- resp_body_json(resp)$data
  data.frame(
    Sepal.Length = sapply(data, `[[`, "Sepal.Length"),
    Sepal.Width = sapply(data, `[[`, "Sepal.Width"),
    Petal.Length = sapply(data, `[[`, "Petal.Length"),
    Petal.Width = sapply(data, `[[`, "Petal.Width"),
    Species = sapply(data, `[[`, "Species")
  )
})
```

`req_perform_parallel` *Perform a list of requests in parallel*

Description

This variation on `req_perform_sequential()` performs multiple requests in parallel. Exercise caution when using this function; it's easy to pummel a server with many simultaneous requests. Only use it with hosts designed to serve many files at once, which are typically web servers, not API servers.

`req_perform_parallel()` has a few limitations:

- Will not retrieve a new OAuth token if it expires part way through the requests.
- Does not perform throttling with `req_throttle()`.
- Does not attempt retries as described by `req_retry()`.
- Only consults the cache set by `req_cache()` before/after all requests.

If any of these limitations are problematic for your use case, we recommend `req_perform_sequential()` instead.

Usage

```
req_perform_parallel(
  reqs,
  paths = NULL,
  pool = NULL,
  on_error = c("stop", "return", "continue"),
  progress = TRUE
)
```

Arguments

`reqs`	A list of requests.
`paths`	An optional list of paths, if you want to download the request bodies to disks. If supplied, must be the same length as `reqs`.
`pool`	Optionally, a curl pool made by `curl::new_pool()`. Supply this if you want to override the defaults for total concurrent connections (100) or concurrent connections per host (6).
`on_error`	What should happen if one of the requests fails? • `stop`, the default: stop iterating with an error. • `return`: stop iterating, returning all the successful responses received so far, as well as an error object for the failed request. • `continue`: continue iterating, recording errors in the result.
`progress`	Display a progress bar? Use `TRUE` to turn on a basic progress bar, use a string to give it a name, or see progress_bars to customise it in other ways.

Value

A list, the same length as `reqs`, containing responses and possibly error objects, if `on_error` is `"return"` or `"continue"` and one of the responses errors. If `on_error` is `"return"` and it errors on the ith request, the ith element of the result will be an error object, and the remaining elements will be `NULL`. If `on_error` is `"continue"`, it will be a mix of requests and error objects.

Only httr2 errors are captured; see `req_error()` for more details.

Examples

```
# Requesting these 4 pages one at a time would take 2 seconds:
request_base <- request(example_url())
reqs <- list(
  request_base |> req_url_path("/delay/0.5"),
  request_base |> req_url_path("/delay/0.5"),
  request_base |> req_url_path("/delay/0.5"),
  request_base |> req_url_path("/delay/0.5")
)
# But it's much faster if you request in parallel
system.time(resps <- req_perform_parallel(reqs))

# req_perform_parallel() will fail on error
reqs <- list(
  request_base |> req_url_path("/status/200"),
  request_base |> req_url_path("/status/400"),
  request("FAILURE")
)
try(resps <- req_perform_parallel(reqs))

# but can use on_error to capture all successful results
resps <- req_perform_parallel(reqs, on_error = "continue")

# Inspect the successful responses
resps |> resps_successes()

# And the failed responses
resps |> resps_failures() |> resps_requests()
```

---

req_perform_sequential

Perform multiple requests in sequence

Description

Given a list of requests, this function performs each in turn, returning a list of responses. It's slower than `req_perform_parallel()` but has fewer limitations.

Usage

```
req_perform_sequential(
  reqs,
  paths = NULL,
  on_error = c("stop", "return", "continue"),
  progress = TRUE
)
```

Arguments

reqs	A list of requests.
paths	An optional list of paths, if you want to download the request bodies to disks. If supplied, must be the same length as reqs.
on_error	What should happen if one of the requests fails? • stop, the default: stop iterating with an error. • return: stop iterating, returning all the successful responses received so far, as well as an error object for the failed request. • continue: continue iterating, recording errors in the result.
progress	Display a progress bar? Use TRUE to turn on a basic progress bar, use a string to give it a name, or see progress_bars to customise it in other ways.

Value

A list, the same length as reqs, containing responses and possibly error objects, if on_error is "return" or "continue" and one of the responses errors. If on_error is "return" and it errors on the ith request, the ith element of the result will be an error object, and the remaining elements will be NULL. If on_error is "continue", it will be a mix of requests and error objects.

Only httr2 errors are captured; see req_error() for more details.

Examples

```
# One use of req_perform_sequential() is if the API allows you to request
# data for multiple objects, you want data for more objects than can fit
# in one request.
req <- request("https://api.restful-api.dev/objects")

# Imagine we have 50 ids:
ids <- sort(sample(100, 50))

# But the API only allows us to request 10 at time. So we first use split
# and some modulo arithmetic magic to generate chunks of length 10
chunks <- unname(split(ids, (seq_along(ids) - 1) %/% 10))

# Then we use lapply to generate one request for each chunk:
reqs <- chunks |> lapply(\(idx) req |> req_url_query(id = idx, .multi = "comma"))

# Then we can perform them all and get the results
## Not run:
```

```
resps <- reqs |> req_perform_sequential()
resps_data(resps, \(resp) resp_body_json(resp))

## End(Not run)
```

`req_perform_stream` *Perform a request and handle data as it streams back*

Description

After preparing a request, call `req_perform_stream()` to perform the request and handle the result with a streaming callback. This is useful for streaming HTTP APIs where potentially the stream never ends.

The `callback` will only be called if the result is successful. If you need to stream an error response, you can use `req_error()` to suppress error handling so that the body is streamed to you.

Usage

```
req_perform_stream(
  req,
  callback,
  timeout_sec = Inf,
  buffer_kb = 64,
  round = c("byte", "line")
)
```

Arguments

`req`	A request.
`callback`	A single argument callback function. It will be called repeatedly with a raw vector whenever there is at least `buffer_kb` worth of data to process. It must return `TRUE` to continue streaming.
`timeout_sec`	Number of seconds to process stream for.
`buffer_kb`	Buffer size, in kilobytes.
`round`	How should the raw vector sent to `callback` be rounded? Choose `"byte"`, `"line"`, or supply your own function that takes a raw vector of `bytes` and returns the locations of possible cut points (or `integer()` if there are none).

Value

An HTTP response. The body will be empty if the request was successful (since the `callback` function will have handled it). The body will contain the HTTP response body if the request was unsuccessful.

Examples

```
show_bytes <- function(x) {
  cat("Got ", length(x), " bytes\n", sep = "")
  TRUE
}
resp <- request(example_url()) |>
  req_url_path("/stream-bytes/100000") |>
  req_perform_stream(show_bytes, buffer_kb = 32)
resp
```

`req_progress` *Add a progress bar to long downloads or uploads*

Description

When uploading or downloading a large file, it's often useful to provide a progress bar so that you know how long you have to wait.

Usage

```
req_progress(req, type = c("down", "up"))
```

Arguments

`req`	A request.
`type`	Type of progress to display: either number of bytes uploaded or downloaded.

Examples

```
req <- request("https://r4ds.s3.us-west-2.amazonaws.com/seattle-library-checkouts.csv") |>
  req_progress()

## Not run:
path <- tempfile()
req |> req_perform(path = path)

## End(Not run)
```

req_proxy *Use a proxy for a request*

Description

Use a proxy for a request

Usage

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

Arguments

req	A request.
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.

Examples

```
# Proxy from https://www.proxynova.com/proxy-server-list/
## Not run:
request("http://hadley.nz") |>
  req_proxy("20.116.130.70", 3128) |>
  req_perform()

## End(Not run)
```

req_retry *Control when a request will retry, and how long it will wait between tries*

Description

`req_retry()` alters `req_perform()` so that it will automatically retry in the case of failure. To activate it, you must specify either the total number of requests to make with `max_tries` or the total amount of time to spend with `max_seconds`. Then `req_perform()` will retry if:

- Either the HTTP request or HTTP response doesn't complete successfully leading to an error from curl, the lower-level library that httr2 uses to perform HTTP request. This occurs, for example, if your wifi is down.
- The error is "transient", i.e. it's an HTTP error that can be resolved by waiting. By default, 429 and 503 statuses are treated as transient, but if the API you are wrapping has other transient status codes (or conveys transient-ness with some other property of the response), you can override the default with `is_transient`.

It's a bad idea to immediately retry a request, so `req_perform()` will wait a little before trying again:

- If the response contains the `Retry-After` header, httr2 will wait the amount of time it specifies. If the API you are wrapping conveys this information with a different header (or other property of the response) you can override the default behaviour with `retry_after`.
- Otherwise, httr2 will use "truncated exponential backoff with full jitter", i.e. it will wait a random amount of time between one second and `2 ^ tries` seconds, capped to at most 60 seconds. In other words, it waits `runif(1, 1, 2)` seconds after the first failure, `runif(1, 1, 4)` after the second, `runif(1, 1, 8)` after the third, and so on. If you'd prefer a different strategy, you can override the default with `backoff`.

Usage

```
req_retry(
  req,
  max_tries = NULL,
  max_seconds = NULL,
  is_transient = NULL,
  backoff = NULL,
  after = NULL
)
```

Arguments

`req`	A request.
`max_tries, max_seconds`	Cap the maximum number of attempts with `max_tries` or the total elapsed time from the first request with `max_seconds`. If neither option is supplied (the default), `req_perform()` will not retry.
`is_transient`	A predicate function that takes a single argument (the response) and returns `TRUE` or `FALSE` specifying whether or not the response represents a transient error.
`backoff`	A function that takes a single argument (the number of failed attempts so far) and returns the number of seconds to wait.

after — A function that takes a single argument (the response) and returns either a number of seconds to wait or NULL, which indicates that a precise wait time is not available that the backoff strategy should be used instead..

Value

A modified HTTP request.

See Also

req_throttle() if the API has a rate-limit but doesn't expose the limits in the response.

Examples

```
# google APIs assume that a 500 is also a transient error
request("http://google.com") |>
  req_retry(is_transient = \(resp) resp_status(resp) %in% c(429, 500, 503))

# use a constant 10s delay after every failure
request("http://example.com") |>
  req_retry(backoff = ~ 10)

# When rate-limited, GitHub's API returns a 403 with
# `X-RateLimit-Remaining: 0` and an Unix time stored in the
# `X-RateLimit-Reset` header. This takes a bit more work to handle:
github_is_transient <- function(resp) {
  resp_status(resp) == 403 &&
    identical(resp_header(resp, "X-RateLimit-Remaining"), "0")
}
github_after <- function(resp) {
  time <- as.numeric(resp_header(resp, "X-RateLimit-Reset"))
  time - unclass(Sys.time())
}
request("http://api.github.com") |>
  req_retry(
    is_transient = github_is_transient,
    after = github_after
  )
```

---

req_template *Set request method/path from a template*

Description

Many APIs document their methods with a lightweight template mechanism that looks like GET /user/{user} or POST /organisation/:org. This function makes it easy to copy and paste such snippets and retrieve template variables either from function arguments or the current environment.

req_template() will append to the existing path so that you can set a base url in the initial request(). This means that you'll generally want to avoid multiple req_template() calls on the same request.

Usage

```
req_template(req, template, ..., .env = parent.frame())
```

Arguments

req	A request.
template	A template string which consists of a optional HTTP method and a path containing variables labelled like either :foo or {foo}.
...	Template variables.
.env	Environment in which to look for template variables not found in .... Expert use only.

Value

A modified HTTP request.

Examples

```
httpbin <- request(example_url())

# You can supply template parameters in `...`
httpbin |> req_template("GET /bytes/{n}", n = 100)

# or you retrieve from the current environment
n <- 200
httpbin |> req_template("GET /bytes/{n}")

# Existing path is preserved:
httpbin_test <- request(example_url()) |> req_url_path("/test")
name <- "id"
value <- "a3fWa"
httpbin_test |> req_template("GET /set/{name}/{value}")
```

---

req_throttle *Rate limit a request by automatically adding a delay*

Description

Use req_throttle() to ensure that repeated calls to req_perform() never exceed a specified rate.

Usage

```
req_throttle(req, rate, realm = NULL)
```

Arguments

req	A request.
rate	Maximum rate, i.e. maximum number of requests per second. Usually easiest expressed as a fraction, number_of_requests / number_of_seconds, e.g. 15 requests per minute is 15 / 60.
realm	A string that uniquely identifies the throttle pool to use (throttling limits always apply *per pool*). If not supplied, defaults to the hostname of the request.

Value

A modified HTTP request.

See Also

req_retry() for another way of handling rate-limited APIs.

Examples

```
# Ensure we never send more than 30 requests a minute
req <- request(example_url()) |>
  req_throttle(rate = 30 / 60)

resp <- req_perform(req)
throttle_status()
resp <- req_perform(req)
throttle_status()
```

---

req_timeout *Set time limit for a request*

Description

An error will be thrown if the request does not complete in the time limit.

Usage

```
req_timeout(req, seconds)
```

Arguments

req	A request.
seconds	Maximum number of seconds to wait

Value

A modified HTTP request.

Examples

```
# Give up after at most 10 seconds
request("http://example.com") |> req_timeout(10)
```

`req_url` *Modify request URL*

Description

- `req_url()` replaces the entire url
- `req_url_query()` modifies the components of the query
- `req_url_path()` modifies the path
- `req_url_path_append()` adds to the path

Usage

```
req_url(req, url)

req_url_query(.req, ..., .multi = c("error", "comma", "pipe", "explode"))

req_url_path(req, ...)

req_url_path_append(req, ...)
```

Arguments

`req`, `.req`	A request.
`url`	New URL; completely replaces existing.
`...`	For `req_url_query()`: <dynamic-dots> Name-value pairs that define query parameters. Each value must be either an atomic vector or `NULL` (which removes the corresponding parameters). If you want to opt out of escaping, wrap strings in `I()`. For `req_url_path()` and `req_url_path_append()`: A sequence of path components that will be combined with `/`.
`.multi`	Controls what happens when an element of `...` is a vector containing multiple values: • `"error"`, the default, throws an error. • `"comma"`, separates values with a `,`, e.g. `?x=1,2`. • `"pipe"`, separates values with a `|`, e.g. `?x=1|2`. • `"explode"`, turns each element into its own parameter, e.g. `?x=1&x=2`. If none of these functions work, you can alternatively supply a function that takes a character vector and returns a string.

Value

A modified HTTP request.

Examples

```
req <- request("http://example.com")

# Change url components
req |>
  req_url_path_append("a") |>
  req_url_path_append("b") |>
  req_url_path_append("search.html") |>
  req_url_query(q = "the cool ice")

# Change complete url
req |>
  req_url("http://google.com")

# Use .multi to control what happens with vector parameters:
req |> req_url_query(id = 100:105, .multi = "comma")
req |> req_url_query(id = 100:105, .multi = "explode")

# If you have query parameters in a list, use !!!
params <- list(a = "1", b = "2")
req |>
  req_url_query(!!!params, c = "3")
```

---

req_user_agent *Set user-agent for a request*

Description

This overrides the default user-agent set by httr2 which includes the version numbers of httr2, the curl package, and libcurl.

Usage

```
req_user_agent(req, string = NULL)
```

Arguments

req	A request.
string	String to be sent in the User-Agent header. If NULL, will user default.

Value

A modified HTTP request.

Examples

```
# Default user-agent:
request("http://example.com") |> req_dry_run()

request("http://example.com") |> req_user_agent("MyString") |> req_dry_run()

# If you're wrapping in an API in a package, it's polite to set the
# user agent to identify your package.
request("http://example.com") |>
  req_user_agent("MyPackage (http://mypackage.com)") |>
  req_dry_run()
```

`req_verbose` *Show extra output when request is performed*

Description

`req_verbose()` uses the following prefixes to distinguish between different components of the HTTP requests and responses:

- `*` informative curl messages
- `->` request headers
- `>>` request body
- `<-` response headers
- `<<` response body

Usage

```
req_verbose(
  req,
  header_req = TRUE,
  header_resp = TRUE,
  body_req = FALSE,
  body_resp = FALSE,
  info = FALSE,
  redact_headers = TRUE
)
```

Arguments

`req`	A request.
`header_req`, `header_resp`	Show request/response headers?
`body_req`, `body_resp`	Should request/response bodies? When the response body is compressed, this will show the number of bytes received in each "chunk".

info	Show informational text from curl? This is mainly useful for debugging https and auth problems, so is disabled by default.
redact_headers	Redact confidential data in the headers? Currently redacts the contents of the Authorization header to prevent you from accidentally leaking credentials when debugging/reprexing.

Value

A modified HTTP request.

See Also

req_perform() which exposes a limited subset of these options through the verbosity argument and with_verbosity() which allows you to control the verbosity of requests deeper within the call stack.

Examples

```
# Use `req_verbose()` to see the headers that are sent back and forth when
# making a request
resp <- request("https://httr2.r-lib.org") |>
  req_verbose() |>
  req_perform()

# Or use one of the convenient shortcuts:
resp <- request("https://httr2.r-lib.org") |>
  req_perform(verbosity = 1)
```

---

resps_successes *Tools for working with lists of responses*

Description

These function provide a basic toolkit for operating with lists of responses and possibly errors, as returned by req_perform_parallel(), req_perform_sequential() and req_perform_iterative().

- resps_successes() returns a list successful responses.
- resps_failures() returns a list failed responses (i.e. errors).
- resps_requests() returns the list of requests that corresponds to each request.
- resps_data() returns all the data in a single vector or data frame. It requires the vctrs package to be installed.

Usage

```
resps_successes(resps)

resps_failures(resps)

resps_requests(resps)

resps_data(resps, resp_data)
```

Arguments

resps	A list of responses (possibly including errors).
resp_data	A function that takes a response (resp) and returns the data found inside that response as a vector or data frame.

Examples

```
reqs <- list(
  request(example_url()) |> req_url_path("/ip"),
  request(example_url()) |> req_url_path("/user-agent"),
  request(example_url()) |> req_template("/status/:status", status = 404),
  request("INVALID")
)
resps <- req_perform_parallel(reqs, on_error = "continue")

# find successful responses
resps |> resps_successes()

# collect all their data
resps |> resps_successes() |> resps_data(\(resp) resp_body_json(resp))

# find requests corresponding to failure responses
resps |> resps_failures() |> resps_requests()
```

---

resp_body_raw *Extract body from response*

Description

- resp_body_raw() returns the raw bytes.
- resp_body_string() returns a UTF-8 string.
- resp_body_json() returns parsed JSON.
- resp_body_html() returns parsed HTML.
- resp_body_xml() returns parsed XML.
- resp_has_body() returns TRUE if the response has a body.

resp_body_json() and resp_body_xml() check that the content-type header is correct; if the server returns an incorrect type you can suppress the check with check_type = FALSE. These two functions also cache the parsed object so the second and subsequent calls are low-cost.

Usage

```
resp_body_raw(resp)

resp_has_body(resp)

resp_body_string(resp, encoding = NULL)

resp_body_json(resp, check_type = TRUE, simplifyVector = FALSE, ...)

resp_body_html(resp, check_type = TRUE, ...)

resp_body_xml(resp, check_type = TRUE, ...)
```

Arguments

resp	A response object.
encoding	Character encoding of the body text. If not specified, will use the encoding specified by the content-type, falling back to UTF-8 with a warning if it cannot be found. The resulting string is always re-encoded to UTF-8.
check_type	Check that response has expected content type? Set to FALSE to suppress the automated check
simplifyVector	Should JSON arrays containing only primitives (i.e. booleans, numbers, and strings) be caused to atomic vectors?
...	Other arguments passed on to jsonlite::fromJSON() and xml2::read_xml() respectively.

Value

- resp_body_raw() returns a raw vector.
- resp_body_string() returns a string.
- resp_body_json() returns NULL, an atomic vector, or list.
- resp_body_html() and resp_body_xml() return an xml2::xml_document

Examples

```
resp <- request("https://httr2.r-lib.org") |> req_perform()
resp

resp |> resp_has_body()
resp |> resp_body_raw()
resp |> resp_body_string()

if (requireNamespace("xml2", quietly = TRUE)) {
  resp |> resp_body_html()
}
```

resp_check_content_type *Check the content type of a response*

Description

A different content type than expected often leads to an error in parsing the response body. This function checks that the content type of the response is as expected and fails otherwise.

Usage

```
resp_check_content_type(
  resp,
  valid_types = NULL,
  valid_suffix = NULL,
  check_type = TRUE,
  call = caller_env()
)
```

Arguments

resp	A response object.
valid_types	A character vector of valid MIME types. Should only be specified with `type/subtype`.
valid_suffix	A string given an "structured media type" suffix.
check_type	Should the type actually be checked? Provided as a convenience for when using this function inside `resp_body_*` helpers.
call	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.

Value

Called for its side-effect; erroring if the response does not have the expected content type.

Examples

```
resp <- response(headers = list(`content-type` = "application/json"))
resp_check_content_type(resp, "application/json")
try(resp_check_content_type(resp, "application/xml"))

# `types` can also specify multiple valid types
resp_check_content_type(resp, c("application/xml", "application/json"))
```

resp_content_type *Extract response content type and encoding*

Description

`resp_content_type()` returns the just the type and subtype of the from the `Content-Type` header. If `Content-Type` is not provided; it returns `NA`. Used by `resp_body_json()`, `resp_body_html()`, and `resp_body_xml()`.

`resp_encoding()` returns the likely character encoding of text types, as parsed from the `charset` parameter of the `Content-Type` header. If that header is not found, not valid, or no charset parameter is found, returns `UTF-8`. Used by `resp_body_string()`.

Usage

```
resp_content_type(resp)

resp_encoding(resp)
```

Arguments

`resp`	An HTTP response object, as created by `req_perform()`.

Value

A string. If no content type is specified `resp_content_type()` will return a character `NA`; if no encoding is specified, `resp_encoding()` will return `"UTF-8"`.

Examples

```
resp <- response(headers = "Content-type: text/html; charset=utf-8")
resp |> resp_content_type()
resp |> resp_encoding()

# No Content-Type header
resp <- response()
resp |> resp_content_type()
resp |> resp_encoding()
```

resp_date *Extract request date from response*

Description

All responses contain a request date in the `Date` header; if not provided by the server will be automatically added by httr2.

Usage

```
resp_date(resp)
```

Arguments

resp	An HTTP response object, as created by req_perform().

Value

A POSIXct date-time.

Examples

```
resp <- response(headers = "Date: Wed, 01 Jan 2020 09:23:15 UTC")
resp |> resp_date()

# If server doesn't add header (unusual), you get the time the request
# was created:
resp <- response()
resp |> resp_date()
```

---

resp_headers *Extract headers from a response*

Description

- resp_headers() retrieves a list of all headers.
- resp_header() retrieves a single header.
- resp_header_exists() checks if a header is present.

Usage

```
resp_headers(resp, filter = NULL)

resp_header(resp, header, default = NULL)

resp_header_exists(resp, header)
```

Arguments

resp	An HTTP response object, as created by req_perform().
filter	A regular expression used to filter the header names. NULL, the default, returns all headers.
header	Header name (case insensitive)
default	Default value to use if header doesn't exist.

Value

- `resp_headers()` returns a list.
- `resp_header()` returns a string if the header exists and `NULL` otherwise.
- `resp_header_exists()` returns `TRUE` or `FALSE`.

Examples

```
resp <- request("https://httr2.r-lib.org") |> req_perform()
resp |> resp_headers()
resp |> resp_headers("x-")

resp |> resp_header_exists("server")
resp |> resp_header("server")
# Headers are case insensitive
resp |> resp_header("SERVER")

# Returns NULL if header doesn't exist
resp |> resp_header("this-header-doesnt-exist")
```

---

resp_link_url *Parse link URL from a response*

Description

Parses URLs out of the the `Link` header as defined by RFC 8288.

Usage

```
resp_link_url(resp, rel)
```

Arguments

`resp`	An HTTP response object, as created by `req_perform()`.
`rel`	The "link relation type" value for which to retrieve a URL.

Value

Either a string providing a URL, if the specified `rel` exists, or `NULL` if not.

Examples

```
# Simulate response from GitHub code search
resp <- response(headers = paste0("Link: ",
  '<https://api.github.com/search/code?q=addClass+user%3Amozilla&page=2>; rel="next",',
  '<https://api.github.com/search/code?q=addClass+user%3Amozilla&page=34>; rel="last"'
))

resp_link_url(resp, "next")
```

```
resp_link_url(resp, "last")
resp_link_url(resp, "prev")
```

`resp_raw` *Show the raw response*

Description

This function reconstructs the HTTP message that httr2 received from the server. It's unlikely to be exactly byte-for-byte identical (because most servers compress at least the body, and HTTP/2 can also compress the headers), but it conveys the same information.

Usage

```
resp_raw(resp)
```

Arguments

`resp`	An HTTP response

Value

`resp` (invisibly).

Examples

```
resp <- request(example_url()) |>
  req_url_path("/json") |>
  req_perform()
resp |> resp_raw()
```

`resp_retry_after` *Extract wait time from a response*

Description

Computes how many seconds you should wait before retrying a request by inspecting the `Retry-After` header. It parses both forms (absolute and relative) and returns the number of seconds to wait. If the heading is not found, it will return `NA`.

Usage

```
resp_retry_after(resp)
```

Arguments

`resp`	An HTTP response object, as created by `req_perform()`.

Value

Scalar double giving the number of seconds to wait before retrying a request.

Examples

```
resp <- response(headers = "Retry-After: 30")
resp |> resp_retry_after()

resp <- response(headers = "Retry-After: Mon, 20 Sep 2025 21:44:05 UTC")
resp |> resp_retry_after()
```

---

resp_status *Extract HTTP status from response*

Description

- `resp_status()` retrieves the numeric HTTP status code
- `resp_status_desc()` retrieves the brief textual description.
- `resp_is_error()` returns `TRUE` if the status code represents an error (i.e. a 4xx or 5xx status).
- `resp_check_status()` turns HTTPs errors into R errors.

These functions are mostly for internal use because in most cases you will only ever see a 200 response:

- 1xx are handled internally by curl.
- 3xx redirects are automatically followed. You will only see them if you have deliberately suppressed redirects with `req |> req_options(followlocation = FALSE)`.
- 4xx client and 5xx server errors are automatically turned into R errors. You can stop them from being turned into R errors with `req_error()`, e.g. `req |> req_error(is_error = ~ FALSE)`.

Usage

```
resp_status(resp)

resp_status_desc(resp)

resp_is_error(resp)

resp_check_status(resp, info = NULL, error_call = caller_env())
```

Arguments

`resp`	An HTTP response object, as created by `req_perform()`.
`info`	A character vector of additional information to include in the error message. Passed to `rlang::abort()`.
`error_call`	The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.

Value

- `resp_status()` returns a scalar integer
- `resp_status_desc()` returns a string
- `resp_is_error()` returns `TRUE` or `FALSE`
- `resp_check_status()` invisibly returns the response if it's ok; otherwise it throws an error with class `httr2_http_{status}`.

Examples

```
# An HTTP status code you're unlikely to see in the wild:
resp <- response(418)
resp |> resp_is_error()
resp |> resp_status()
resp |> resp_status_desc()
```

---

resp_url *Get URL/components from the response*

Description

- `resp_url()` returns the complete url.
- `resp_url_path()` returns the path component.
- `resp_url_query()` returns a single query component.
- `resp_url_queries()` returns the query component as a named list.

Usage

```
resp_url(resp)

resp_url_path(resp)

resp_url_query(resp, name, default = NULL)

resp_url_queries(resp)
```

Arguments

`resp`	An HTTP response object, as created by `req_perform()`.
`name`	Query parameter name.
`default`	Default value to use if query parameter doesn't exist.

Examples

```
resp <- request(example_url()) |>
  req_url_path("/get?hello=world") |>
  req_perform()

resp |> resp_url()
resp |> resp_url_path()
resp |> resp_url_queries()
resp |> resp_url_query("hello")
```

`secrets` *Secret management*

Description

httr2 provides a handful of functions designed for working with confidential data. These are useful because testing packages that use httr2 often requires some confidential data that needs to be available for testing, but should not be available to package users.

- `secret_encrypt()` and `secret_decrypt()` work with individual strings
- `secret_encrypt_file()` encrypts a file in place and `secret_decrypt_file()` decrypts a file in a temporary location.
- `secret_write_rds()` and `secret_read_rds()` work with `.rds` files
- `secret_make_key()` generates a random string to use as a key.
- `secret_has_key()` returns `TRUE` if the key is available; you can use it in examples and vignettes that you want to evaluate on your CI, but not for CRAN/package users.

These all look for the key in an environment variable. When used inside of testthat, they will automatically `testthat::skip()` the test if the env var isn't found. (Outside of testthat, they'll error if the env var isn't found.)

Usage

```
secret_make_key()

secret_encrypt(x, key)

secret_decrypt(encrypted, key)

secret_write_rds(x, path, key)

secret_read_rds(path, key)

secret_decrypt_file(path, key, envir = parent.frame())

secret_encrypt_file(path, key)

secret_has_key(key)
```

Arguments

x	Object to encrypt. Must be a string for secret_encrypt().
key	Encryption key; this is the password that allows you to "lock" and "unlock" the secret. The easiest way to specify this is as the name of an environment variable. Alternatively, if you already have a base64url encoded string, you can wrap it in I(), or you can pass the raw vector in directly.
encrypted	String to decrypt
path	Path to .rds file
envir	The decrypted file will be automatically deleted when this environment exits. You should only need to set this argument if you want to pass the unencrypted file to another function.

Value

- secret_decrypt() and secret_encrypt() return strings.
- secret_write_rds() returns x invisibly; secret_read_rds() returns the saved object.
- secret_make_key() returns a string with class AsIs.
- secret_has_key() returns TRUE or FALSE.

Basic workflow

1. Use secret_make_key() to generate a password. Make this available as an env var (e.g. {MYPACKAGE}_KEY) by adding a line to your .Renviron.
2. Encrypt strings with secret_encrypt(), files with secret_encrypt_file(), and other data with secret_write_rds(), setting key = "{MYPACKAGE}_KEY".
3. In your tests, decrypt the data with secret_decrypt(), secret_decrypt_file(), or secret_read_rds() to match how you encrypt it.
4. If you push this code to your CI server, it will already "work" because all functions automatically skip tests when your {MYPACKAGE}_KEY env var isn't set. To make the tests actually run, you'll need to set the env var using whatever tool your CI system provides for setting env vars. Make sure to carefully inspect the test output to check that the skips have actually gone away.

Examples

```
key <- secret_make_key()

path <- tempfile()
secret_write_rds(mtcars, path, key = key)
secret_read_rds(path, key)

# While you can manage the key explicitly in a variable, it's much
# easier to store in an environment variable. In real life, you should
# NEVER use `Sys.setenv()` to create this env var because you will
# also store the secret in your `.Rhistory`. Instead add it to your
# .Renviron using `usethis::edit_r_environ()` or similar.
Sys.setenv("MY_KEY" = key)
```

```
x <- secret_encrypt("This is a secret", "MY_KEY")
x
secret_decrypt(x, "MY_KEY")
```

`url_parse` *Parse and build URLs*

Description

`url_parse()` parses a URL into its component pieces; `url_build()` does the reverse, converting a list of pieces into a string URL. See RFC 3986 for the details of the parsing algorithm.

Usage

```
url_parse(url)

url_build(url)
```

Arguments

`url`	For `url_parse()` a string to parse into a URL; for `url_build()` a URL to turn back into a string.

Value

- `url_build()` returns a string.
- `url_parse()` returns a URL: a S3 list with class `httr2_url` and elements `scheme`, `hostname`, `port`, `path`, `fragment`, `query`, `username`, `password`.

Examples

```
url_parse("http://google.com/")
url_parse("http://google.com:80/")
url_parse("http://google.com:80/?a=1&b=2")
url_parse("http://username@google.com:80/path;test?a=1&b=2#40")

url <- url_parse("http://google.com/")
url$port <- 80
url$hostname <- "example.com"
url$query <- list(a = 1, b = 2, c = 3)
url_build(url)
```

`with_mocked_responses` *Temporarily mock requests*

Description

Mocking allows you to selectively and temporarily replace the response you would typically receive from a request with your own code. It's primarily used for testing.

Usage

```
with_mocked_responses(mock, code)

local_mocked_responses(mock, env = caller_env())
```

Arguments

`mock`	A function, a list, or `NULL`. • `NULL` disables mocking and returns httr2 to regular operation. • A list of responses will be returned in sequence. After all responses have been used up, will return 503 server errors. • For maximum flexibility, you can supply a function that that takes a single argument, `req`, and returns either `NULL` (if it doesn't want to handle the request) or a response (if it does).
`code`	Code to execute in the temporary environment.
`env`	Environment to use for scoping changes.

Value

`with_mock()` returns the result of evaluating `code`.

Examples

```
# This function should perform a response against google.com:
google <- function() {
  request("http://google.com") |>
    req_perform()
}

# But I can use a mock to instead return my own made up response:
my_mock <- function(req) {
  response(status_code = 403)
}
try(with_mock(my_mock, google()))
```

`with_verbosity` *Temporarily set verbosity for all requests*

Description

`with_verbosity()` is useful for debugging httr2 code buried deep inside another package because it allows you to see exactly what's been sent and requested.

Usage

```
with_verbosity(code, verbosity = 1)
```

Arguments

`code`	Code to execture
`verbosity`	How much information to print? This is a wrapper around `req_verbose()` that uses an integer to control verbosity: • `0`: no output • `1`: show headers • `2`: show headers and bodies • `3`: show headers, bodies, and curl status messages. Use `with_verbosity()` to control the verbosity of requests that you can't affect directly.

Value

The result of evaluating `code`.

Examples

```
fun <- function() {
  request("https://httr2.r-lib.org") |> req_perform()
}
with_verbosity(fun())
```

Index