Package 'httr2'

May 11, 2022

Title Perform HTTP Requests and Process the Responses

Version 0.2.1

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 (>= 3.4)

Imports cli (>= 3.0.0), curl, glue, magrittr, openssl, R6, rappdirs, rlang (>= 1.0.0), withr

Suggests askpass, bench, clipr, covr, docopt, httpuv, jose, jsonlite, knitr, purrr, rmarkdown, testthat (>= 3.0.0), tibble, webfakes, xml2

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.1.2

NeedsCompilation no

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

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2022-05-10 22:30:02 UTC

R topics documented:

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)

curl_help()
```

Arguments

`cmd`	Call to curl. If omitted and the clipr package is installed, will be retrieved from the clipboard.

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")
```

jwt_claim *Create and encode a JWT*

Description

`jwt_claim()` is a wrapper around `jose::jwt_claim()` that creates a JWT claim set with a few extra default values. `jwt_encode_sig()` and `jwt_encode_hmac()` are thin wrappers around `jose::jwt_encode_sig()` and `jose::jwt_encode_hmac()` that exist primarily to make specification in other functions a little simpler.

Usage

```
jwt_claim(
  iss = NULL,
  sub = NULL,
  aud = NULL,
  exp = unix_time() + 5L * 60L,
  nbf = unix_time(),
  iat = unix_time(),
  jti = NULL,
  ...
)

jwt_encode_sig(claim, key, size = 256, header = list())

jwt_encode_hmac(claim, secret, size = size, header = list())
```

Arguments

iss	Issuer claim. Identifies the principal that issued the JWT.
sub	Subject claim. Identifies the principal that is the subject of the JWT (i.e. the entity that the claims apply to).
aud	Audience claim. Identifies the recipients that the JWT is intended. Each principle intended to process the JWT must be identifid with a unique value.
exp	Expiration claim. Identifies the expiration time on or after which the JWT MUST NOT be accepted for processing. Defaults to 5 minutes.
nbf	Not before claim. Identifies the time before which the JWT MUST NOT be accepted for processing. Defaults to current time.
iat	Issued at claim. Identifies the time at which the JWT was issued. Defaults to current time.
jti	JWT ID claim. Provides a unique identifier for the JWT. If omitted, uses a random 32-byte sequence encoded with base64url.
...	Any additional claims to include in the claim set.
claim	Claim set produced by `jwt_claim()`.
key	RSA or EC private key either specified as a path to a file, a connection, or a string (PEM/SSH format), or a raw vector (DER format).
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.
secret	String or raw vector with a secret passphrase.

Value

An S3 list with class `jwt_claim`.

Examples

```
claim <- jwt_claim()
str(claim)
```

---

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()
```

---

multi_req_perform *Perform multiple requests in parallel*

Description

This variation on `req_perform()` performs multiple requests in parallel. Unlike `req_perform()` it always succeeds; it will never throw an error. Instead it will return error objects, which are your responsibility to handle.

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.

Usage

```
multi_req_perform(reqs, paths = NULL, pool = NULL, cancel_on_error = FALSE)
```

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).
cancel_on_error	Should all pending requests be cancelled when you hit an error. Set this to TRUE to stop all requests as soon as you hit an error. Responses that were never performed will have class httr2_cancelled in the result.

Value

A list the same length as reqs where each element is either a response or an error.

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().
- Consults the cache set by req_cache() before/after all requests.

In general, where req_perform() might make multiple requests due to retries or OAuth failures, multi_req_perform() will make only make 1.

Examples

```
# Requesting these 4 pages one at a time would take four seconds:
reqs <- list(
  request("https://httpbin.org/delay/1"),
  request("https://httpbin.org/delay/1"),
  request("https://httpbin.org/delay/1"),
  request("https://httpbin.org/delay/1")
)
# But it's much faster if you request in parallel
system.time(resps <- multi_req_perform(reqs))

reqs <- list(
  request("https://httpbin.org/status/200"),
  request("https://httpbin.org/status/400"),
  request("FAILURE")
)
# multi_req_perform() will always succeed
resps <- multi_req_perform(reqs)
# you'll need to inspect the results to figure out which requests fails
fail <- vapply(resps, inherits, "error", FUN.VALUE = logical(1))
resps[fail]
```

`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 recorded 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 in `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 cache directory. If `NULL`, generated from hash of `client_id`. If you're defining a package 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 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 rfc6749, Section 2.3.1.
- `oauth_client_req_header()` adds the client id and secret using HTTP basic authentication with the `Authorization` header, as described in rfc6749, Section 2.3.1.
- `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 rfc7523, Section 2.2.

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_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.

Examples

```
oauth_token("abcdef")
oauth_token("abcdef", expires_in = Sys.time() + 3600)
oauth_token("abcdef", refresh_token = "ghijkl")
```

---

obfuscate *Obfuscate mildly secret information*

Description

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

Because un-obfuscation happens at the last possible instant, obfuscated() only works 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().

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 response: 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

RFC750 The OAuth 2.0 Authorization Framework: Bearer Token Usage

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.
- `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", ...)

req_body_form(.req, ...)

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

Arguments

req	A request.
body	A literal string or raw vector to send as body.
type	Content type. For `req_body_file()`, the default will will attempt to guess from the extension of `path`.
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"`).
...	Name-data pairs used send data in the body. For `req_body_form()`, the values must be strings (or things easily coerced to string); for `req_body_multipart()` the values must be strings or objects produced by `curl::form_file()`/`curl::form_data()`. For `req_body_json()`, additional arguments passed on to `jsonlite::toJSON()`.
.req	A request.

Value

A modified HTTP request.

Examples

```
req <- request("http://httpbin.org/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 = FALSE)
```

Arguments

`req`	A request.
`path`	Path to cache directory
`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.

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_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 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.

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("http://httpbin.org/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 = function(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 = function(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, ...)
```

Arguments

`.req`	A request.
`...`	Name-value pairs of headers and their values. • Use `NULL` to reset a value to httr's default • Use `""` to remove a header • Use a character vector to repeat a header.

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()
```

---

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("http://httpbin.org") %>% req_method("PATCH")
request("http://httpbin.org") %>% req_method("PUT")
request("http://httpbin.org") %>% req_method("HEAD")
```

---

req_oauth_auth_code *OAuth authentication with authorization code*

Description

This uses `oauth_flow_auth_code()` to generate an access token, which is then used to authentication the request with `req_auth_bearer_token()`. The token is automatically cached (either in memory or on disk) to minimise the number of times the flow is performed.

Usage

```
req_oauth_auth_code(
  req,
  client,
  auth_url,
  cache_disk = FALSE,
  cache_key = NULL,
  scope = NULL,
  pkce = TRUE,
  auth_params = list(),
  token_params = list(),
  host_name = "localhost",
  host_ip = "127.0.0.1",
  port = httpuv::randomPort()
)
```

Arguments

req	A request.
client	An `oauth_client()`.
auth_url	Authorization url; you'll need to discover this by reading the documentation.
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. Cached tokens are encrypted and automatically deleted 30 days after creation.
cache_key	If you want to cache multiple tokens per app, use this key to disambiguate them.
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	List containing additional parameters passed to `oauth_flow_auth_code_url()`
token_params	List containing additional parameters passed to the `token_url`.
host_name	Host name used to generate `redirect_uri`
host_ip	IP address web server will be bound to.
port	Port to bind web server to. By default, this uses a random port. You may need to set it to a fixed port if the API requires that the `redirect_uri` specified in the client exactly matches the `redirect_uri` generated by this function.

Value

A modified HTTP request.

Security considerations

The authorization code flow is used for both web applications and native applications (which are equivalent to R packages). rfc8252 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.

Examples

```
client <- oauth_client(
  id = "28acfec0674bb3da9f38",
  secret = obfuscated(paste0(
     "J9iiGmyelHltyxqrHXW41ZZPZamyUNxSX1_uKnv",
     "PeinhhxET_7FfUs2X0LLKotXY2bpgOMoHRCo"
  )),
  token_url = "https://github.com/login/oauth/access_token",
  name = "hadley-oauth-test"
)

request("https://api.github.com/user") %>%
  req_oauth_auth_code(client, auth_url = "https://github.com/login/oauth/authorize")
```

`req_oauth_bearer_jwt` *OAuth authentication with a bearer JWT*

Description

This uses `oauth_flow_bearer_jwt()` to generate an access token which is then used to authenticate the request with `req_auth_bearer_token()`. The token is cached in memory.

Usage

```
req_oauth_bearer_jwt(
  req,
  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

A modified HTTP request.

Examples

```
client <- oauth_client("example", "https://example.com/get_token")
claim <- jwt_claim()
req <- request("https://example.com")

req %>% req_oauth_bearer_jwt(client, claim)
```

req_oauth_client_credentials *OAuth authentication with client credentials*

Description

This uses `oauth_flow_client_credentials()` to generate an access token, which is then used to authentication the request with `req_auth_bearer_token()`. The token is cached in memory.

Usage

```
req_oauth_client_credentials(req, 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

A modified HTTP request.

Examples

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

req %>% req_oauth_client_credentials(client)
```

req_oauth_device *OAuth authentication with device flow*

Description

This uses `oauth_flow_device()` to generate an access token, which is then used to authentication the request with `req_auth_bearer_token()`. The token is automatically cached (either in memory or on disk) to minimise the number of times the flow is performed.

Usage

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

Arguments

req	A request.
client	An oauth_client().
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. Cached tokens are encrypted and automatically deleted 30 days after creation.
cache_key	If you want to cache multiple tokens per app, use this key to disambiguate them.
scope	Scopes to be requested from the resource owner.
auth_params	List containing additional parameters passed to oauth_flow_auth_code_url()
token_params	List containing additional parameters passed to the token_url.

Value

A modified HTTP request.

Examples

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

req %>% req_oauth_device(client)
```

---

req_oauth_password *OAuth authentication with username and password*

Description

This uses oauth_flow_password() to generate an access token, which is then used to authentication the request with req_auth_bearer_token(). The token, not the password is automatically cached (either in memory or on disk); the password is used once to get the token and is then discarded.

Usage

```
req_oauth_password(
  req,
  client,
  username,
  password = NULL,
  cache_disk = FALSE,
  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.
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. Cached tokens are encrypted and automatically deleted 30 days after creation.
scope	Scopes to be requested from the resource owner.
token_params	List containing additional parameters passed to the token_url.

Value

A modified HTTP request.

Examples

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

if (interactive()) {
  req %>% req_oauth_password(client, "username")
}
```

---

req_oauth_refresh *OAuth authentication with a refresh token*

Description

This uses `oauth_flow_refresh()` to generate an access token, which is then used to authentication the request with `req_auth_bearer_token()`. This is primarily useful for testing: you can manually execute another OAuth flow (e.g. by calling `oauth_flow_auth_code()` or `oauth_flow_device()`), extract the refresh token from the result, and then save in an environment variable for future 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 error, and you'll have to create a new refresh token following the same procedure you did to get the first token (so it's a good idea to document what you did the first time because you might need to do it again).

Usage

```
req_oauth_refresh(
  req,
  client,
  refresh_token = Sys.getenv("HTTR_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 `HTTR_REFRESH_TOKEN`.
`scope`	Scopes to be requested from the resource owner.
`token_params`	List containing additional parameters passed to the `token_url`.

Value

A modified HTTP request.

Examples

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

`req_perform` *Perform a request*

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)
)
```

Arguments

`req`	A request.
`path`	Optionally, path to save body of request. 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.

Value

If request is successful (i.e. the request was successfully performed and a response with HTTP status code <400 was recieved), an HTTP response; otherwise throws an error. Override this behaviour with `req_error()`.

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.

Examples

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

---

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:

- The either the HTTP request or HTTP response doesn't complete successfully leading to an error from curl, the lower-level library that httr 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_status(.x) %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_stream *Perform a request, streaming data back to R*

Description

After preparing a request, call `req_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.

Usage

```
req_stream(req, callback, timeout_sec = Inf, buffer_kb = 64)
```

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 processs stream for.
`buffer_kb`	Buffer size, in kilobytes.

Value

An HTTP response.

Examples

```
show_bytes <- function(x) {
  cat("Got ", length(x), " bytes\n", sep = "")
  TRUE
}
resp <- request("http://httpbin.org/stream-bytes/100000") %>%
  req_stream(show_bytes, buffer_kb = 32)
```

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("http://httpbin.org")

# 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_cookies <- request("http://httpbin.org/cookies")
name <- "id"
value <- "a3fWa"
httpbin_cookies %>% req_template("GET /set/{name}/{value}")
```

---

req_throttle *Throttle 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	An unique identifier that for throttle 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 server will never recieve more than 10 requests a minute
request("https://example.com") %>%
  req_throttle(rate = 10 / 60)
```

---

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, ...)

req_url_path(req, ...)

req_url_path_append(req, ...)
```

Arguments

`req`	A request.
`url`	New URL; completely replaces existing.
`.req`	A request.
`...`	For `req_url_query()`: Name-value pairs that provide query parameters. Each value must be either length-1 atomic vector or `NULL` (which is automatically dropped). For `req_url_path()` and `req_url_path_append()`: A sequence of path components that will be combined with `/`.

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")
```

`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)
```

---

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_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.

Usage

```
resp_body_raw(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_body_raw()
resp %>% resp_body_string()

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

---

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(header = "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)

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)

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 rfc8288.

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("https://httpbin.org/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)
```

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()`.

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()
```

---

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_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_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

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() and other data with secret_write_rds(), setting key = "{MYPACKAGE}_KEY".
3. In your tests, decrypt the data with secret_decrypt() 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 rfc3986 for details of 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_mock *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_mock(mock, code)

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

Arguments

mock	A single argument function called with a request object. It should return 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)
}
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