Package 'biohttp'

September 3, 2026

Title Normalized HTTP Transport with Circuit Breaking and Caching

Version 0.1.2

Description Web service calls return a normalized result value instead of raising a condition, so a caller branches on data rather than on an error handler. Transport failure, a non-success status code, and an unreadable response body are reported as three distinct outcomes. Per-host circuit breaking, retry with a transient-failure predicate, optional throttling, redacted request headers, and a success-only cache come as defaults. Many questions to one source can be asked as a single batch, where only the entries the cache is missing reach the network. Service-specific knowledge is left to the client packages built on top. The circuit breaker is the pattern described in Nygard (2018, ISBN:9781680502398).

License MIT + file LICENSE

URL https://github.com/samuelbharti/biohttp, https://www.samuelbharti.com/biohttp/

BugReports https://github.com/samuelbharti/biohttp/issues

Encoding UTF-8

Language en-US

Depends R (>= 4.0)

Imports cachem, curl, httr2, jsonlite, rlang, tools

Suggests knitr, rmarkdown, testthat (>= 3.0.0), webfakes, withr

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Samuel Bharti [aut, cre, cph] (ORCID: <https://orcid.org/0000-0003-4190-7058>)

Maintainer Samuel Bharti <samuelbharti.io@gmail.com>

Repository CRAN

Date/Publication 2026-09-03 11:50:41 UTC

Contents

as_legacy_envelope . . . 2
body_or_null . . . 3
breaker_open . . . 4
breaker_record . . . 5
breaker_reset . . . 5
cache . . . 6
cached . . . 7
cache_key . . . 8
cache_reset . . . 9
classify_condition . . . 9
classify_http . . . 10
envelope . . . 11
get_json . . . 12
get_json_many . . . 13
get_text . . . 15
graphql_error . . . 16
http_error_message . . . 17
is_blank . . . 18
is_transient . . . 18
perform . . . 19
perform_many . . . 20
perform_text . . . 22
pluck_at . . . 23
post_json . . . 23
post_json_many . . . 25
redact_secrets . . . 26
req_defaults . . . 27
status_constructors . . . 28
STATUS_LEVELS . . . 29
status_message . . . 30
user_agent . . . 31

Index **33**

`as_legacy_envelope` *Convert an envelope to the old four-field shape*

Description

Deprecated. Scheduled for removal in 0.3.0.

Translates an `envelope()` into the four-field shape a hand-rolled HTTP layer commonly returns:

```
list(ok = TRUE, status = 200L, data = <parsed>, error = NULL)
```

`detail` is carried through as a fifth field, because call sites that log it would otherwise break on adoption.

Usage

```
as_legacy_envelope(res)
```

Arguments

res	An envelope from perform() or one of the convenience wrappers.

Value

A list with ok, status (the HTTP code), data, error, and detail.

Why this is deprecated on arrival

The old shape cannot express two things the new one can. It has no way to say "skipped because the host's breaker was open" as distinct from "failed", and it collapses the user-facing sentence and the log detail into one field. Both are load-bearing, and neither can be reconstructed once a call site is written against the old shape.

Note the collision in the word status: here it is the HTTP code, an integer, and in an envelope() it is the enum. That is exactly the ambiguity the new contract removes.

Removal

Scheduled for removal in 0.3.0. Use it to land a migration with a reviewable diff, then delete the call sites and delete this.

Warns once per session, not once per call, so a migration loop stays readable.

Examples

```
old <- as_legacy_envelope(status_ok(data = list(n = 1), source = "MyGene"))
old$ok
old$status
```

---

body_or_null *The parsed body, or NULL*

Description

For a client that wants the body and does not care why a call failed. Every other field is still there for a client that does care, so reach for this when the failure is genuinely not actionable, not to avoid reading the envelope.

Usage

```
body_or_null(res)
```

Arguments

res	An envelope from perform() or one of the convenience wrappers.

Details

Returns the body for ok **and for** stale. A stale envelope carries real data that is merely past its freshness window, so returning NULL for it would throw away the one thing the caller asked for. Branch on res$status if the difference matters. Note that res$ok is FALSE for stale, which is what keeps cached() from storing it.

Value

res$data when the status is ok or stale, otherwise NULL.

Examples

```
body_or_null(status_ok(data = list(n = 1)))
body_or_null(status_stale(data = list(n = 1)))
body_or_null(status_error(source = "gnomAD"))
```

---

breaker_open *Is a host's breaker open*

Description

Open means too many consecutive transport failures and still inside the cooldown. While open, perform() short-circuits to a skipped envelope instead of waiting on a request that is likely to fail anyway.

Usage

```
breaker_open(host, now = as.numeric(Sys.time()))
```

Arguments

host	A hostname, as returned by parsing a request URL.
now	The current time in seconds since the epoch. Exposed so a test can drive the clock rather than sleep.

Value

A single logical.

Examples

```
breaker_reset()
breaker_open("gnomad.broadinstitute.org")
```

breaker_record *Record a call outcome against a host*

Description

`reachable` means an HTTP response came back at all, whatever its status. A reachable call clears the count; the Kth consecutive unreachable one trips the breaker for the cooldown window.

Usage

```
breaker_record(host, reachable, now = as.numeric(Sys.time()))
```

Arguments

`host`	A hostname.
`reachable`	Whether an HTTP response arrived.
`now`	The current time in seconds since the epoch.

Details

Pass `reachable = TRUE` for any response in hand, including a 5xx and including a 2xx whose body will not parse. Only a genuine transport failure is `FALSE`.

Value

The host's new breaker state, invisibly.

Examples

```
breaker_reset()
breaker_record("example.org", reachable = FALSE)
breaker_record("example.org", reachable = TRUE)
breaker_open("example.org")
```

breaker_reset *Clear all breaker state*

Description

Forgets every host's failure count and cooldown. For tests, and for a manual "try again" once a source is known to be back.

Usage

```
breaker_reset()
```

Value

NULL, invisibly.

Examples

```
breaker_reset()
```

---

cache *The cache store*

Description

The cachem object backing cached(). Memory-only unless BIOHTTP_CACHE_DISK asks for the disk tier, in which case the two are layered. Built on first use.

Usage

```
cache()
```

Details

It is safe to hold the returned object. cache_reset() clears it in place rather than replacing it, unless a cache setting has changed, in which case a differently configured store genuinely has to be built. See cache_reset().

Value

A cachem cache object.

Examples

```
cache_reset()
class(cache())[1]
```

cached *Serve from cache, or fetch and cache a success*

Description

Returns a cached result for `key` if there is one, otherwise runs `fetch()` and stores the result only when it succeeded.

Usage

```
cached(key, fetch)
```

Arguments

`key`	A key from `cache_key()`.
`fetch`	A function of no arguments returning an envelope.

Details

A failure is never cached. This is the single most important rule in the package. A cache that stores an error fallback poisons itself for the life of the R process, and every later lookup then serves the stored failure instead of retrying. Storing only successes means a transient outage resolves itself the moment the source comes back.

Value

The envelope, from the cache or from `fetch()`.

Examples

```
cache_reset()
key <- cache_key("demo", "GET /x")
cached(key, function() status_ok(data = list(n = 1), source = "demo"))

# A failure runs fetch() again every time.
bad <- cache_key("demo", "GET /y")
cached(bad, function() status_error(source = "demo"))
```

cache_key *Build a cache key*

Description

A stable lowercase-hex hash. cachem keys may contain only lowercase letters and numbers, so a URL cannot be one directly and the inputs are hashed instead. rlang::hash() returns lowercase hex, which satisfies the rule.

Usage

```
cache_key(source, key, params = NULL)
```

Arguments

source	A friendly label for the service.
key	Something identifying the call, usually the method and URL.
params	Anything else that changes the answer, such as a POST body.

Details

The salt and the schema version are inside the hash, so changing either yields a fresh keyspace. Set BIOHTTP_CACHE_SALT per deployment: a shared disk cache would otherwise collide across app versions, and it would leak which queries were run to anyone able to probe it.

Put anything that changes the answer into params**, credentials included.** The wrappers pass their headers through, because two callers hitting the same URL with different tokens can legitimately get different responses, and keying on the URL alone would serve one caller's data to the other. The cost is that rotating a token misses the cache once, which is the right trade.

Value

A single string.

Examples

```
cache_key("gnomAD", "GET https://example.org/v1/gene/BRCA1")
cache_key("gnomAD", "POST https://example.org/graphql", list(q = "BRCA1"))
```

cache_reset *Empty the cache*

Description

Removes every entry. Which of two things happens underneath depends on whether the cache settings have changed since the store was built:

Usage

```
cache_reset()
```

Details

Settings unchanged The existing store is cleared in place, so a reference taken from `cache()` earlier stays valid and keeps pointing at the live cache. This is the ordinary case.

Settings changed The store is dropped so the next `cache()` builds one from the new settings. Tests rely on this after pointing `BIOHTTP_CACHE_DIR` at a fresh tempdir or toggling `BIOHTTP_CACHE_DISK`.

The in-place branch exists because dropping unconditionally silently orphans any held reference: writes through it go somewhere nothing else can see, and the only symptom is a hit rate quietly falling to zero. A reference held across a genuine settings change is still orphaned, but nothing changes cache settings mid-run except a test suite, and those do not hold references.

Value

`NULL`, invisibly.

Examples

```
held <- cache()
cache_reset()
identical(held, cache())
```

classify_condition *Classify a transport condition onto the status enum*

Description

For a failure with no HTTP response at all. A DNS failure or a refused connection is an `error`; only an actual timeout is a `timeout`, because the two read very differently to a user deciding whether to retry.

Usage

```
classify_condition(cond)
```

Arguments

cond	A condition caught from a failed request.

Value

One of STATUS_LEVELS.

Examples

```
classify_condition(simpleError("Timeout was reached"))
classify_condition(simpleError("Could not resolve host"))
```

---

classify_http *Classify an HTTP status code onto the status enum*

Description

Only the codes the transport treats as an outcome in themselves are classified. A 2xx is the caller's to interpret, since a 200 carrying an empty result may still be no_data once parsed.

Usage

```
classify_http(http)
```

Arguments

http	An HTTP status code.

Value

One of STATUS_LEVELS.

Examples

```
classify_http(404L)
classify_http(429L)
classify_http(503L)
```

envelope *Build a result envelope*

Description

The single return shape for the whole package. ok is derived from status rather than passed in, so the two can never disagree.

Usage

```
envelope(
  status,
  data = NULL,
  source = "API",
  http = NA_integer_,
  error = NULL,
  detail = NULL
)
```

Arguments

status	One of STATUS_LEVELS.
data	The parsed body on success, NULL otherwise.
source	A friendly label for the service, used in the user-facing message. For example "gnomAD".
http	The HTTP status code, or NA_integer_ when no response arrived.
error	One sentence fit to show a user. Never carries technical detail.
detail	The technical cause, for a log. Never shown to a user.

Details

A client that only wants the parsed body can use body_or_null() instead of reading the fields directly.

Value

A list with ok, status, http, data, source, error, detail, and ts.

Examples

```
envelope("ok", data = list(symbol = "BRCA1"), source = "MyGene", http = 200L)
envelope("error", source = "gnomAD", http = 503L)
```

`get_json` *GET a JSON endpoint*

Description

Assembles the request, applies `req_defaults()`, performs it, and caches the result if it succeeded.

Usage

```
get_json(
  base_url,
  path = NULL,
  query = list(),
  source = "API",
  timeout = 15,
  max_tries = 3,
  headers = NULL,
  throttle = NULL,
  secret_query = NULL
)
```

Arguments

`base_url`	The service's base URL.
`path`	An optional path appended to `base_url`.
`query`	A named list of query parameters. Blank values are dropped.
`source`	A friendly label for the service.
`timeout`	Seconds before the request is abandoned.
`max_tries`	Total attempts, including the first.
`headers`	A named list of headers, all marked sensitive.
`throttle`	An optional throttle spec. See `req_defaults()`.
`secret_query`	A named list of query-string credentials, for a service that has no header form. Deliberately **not** part of the cache key, so it suits a credential that raises a rate limit and not one that changes the response. See `redact_secrets()`.

Value

An envelope. See `envelope()`.

Examples

```
# Mocked so the example runs offline. A real call drops the
# with_mocked_responses() wrapper.
cache_reset()
breaker_reset()
```

```
httr2::with_mocked_responses(
  list(httr2::response(
    status_code = 200,
    headers = list(`content-type` = "application/json"),
    body = charToRaw('{"hits":[{"symbol":"BRCA1"}]}')
  )),
  get_json(
    "https://mygene.info/v3",
    path = "query",
    query = list(q = "BRCA1", species = "human"),
    source = "MyGene"
  )$data$hits[[1]]$symbol
)
```

`get_json_many` *GET many JSON endpoints as one batch*

Description

The batched counterpart to `get_json()`. Assembles one request per entry in `queries`, serves whatever the cache already holds, performs only the rest, and caches the successes.

Usage

```
get_json_many(
  base_url,
  path = NULL,
  queries = list(),
  source = "API",
  timeout = 15,
  max_tries = 3,
  headers = NULL,
  throttle = NULL,
  max_active = 6,
  progress = FALSE,
  secret_query = NULL
)
```

Arguments

`base_url`	The service's base URL.
`path`	A path appended to `base_url`. Length 1 to use the same path for every request, or one per entry in `queries`.
`queries`	A list of named lists, one per request. Blank values are dropped from each.
`source`	A friendly label for the service, used in the user-facing message. For example `"MyGene"`.

timeout	Seconds before a request is abandoned.
max_tries	Total attempts, including the first. Note that httr2 does not honor this under parallel performance.
headers	A named list of headers, all marked sensitive.
throttle	A throttle spec. See req_defaults().
max_active	Maximum requests in flight at once.
progress	Passed to httr2::req_perform_parallel(). FALSE by default, because the usual caller is a Shiny app that renders its own.
secret_query	A named list of query-string credentials, applied to every request in the batch at dispatch. See redact_secrets().

Details

Read the sections on perform_many() first. The same two rules apply: pass requests for one host, and supply a throttle.

Value

A list of envelopes, the same length and order as queries.

Examples

```
cache_reset()
breaker_reset()

httr2::with_mocked_responses(
  function(req) {
    httr2::response(
      status_code = 200,
      headers = list(`content-type` = "application/json"),
      body = charToRaw('{"hits":[]}')
    )
  },
  length(get_json_many(
    "https://mygene.info/v3",
    path = "query",
    queries = list(list(q = "BRCA1"), list(q = "TP53")),
    source = "MyGene"
  ))
)
```

`get_text` *GET a text endpoint*

Description

For a source that ships a bulk flat file, CSV or TSV, rather than a per-record JSON API. Same timeout, retry, caching, and header redaction as `get_json()`; the body comes back verbatim as a string in `data`.

Usage

```
get_text(
  base_url,
  path = NULL,
  query = list(),
  source = "API",
  timeout = 30,
  max_tries = 3,
  headers = NULL,
  throttle = NULL,
  secret_query = NULL
)
```

Arguments

`base_url`	The service's base URL.
`path`	An optional path appended to `base_url`.
`query`	A named list of query parameters. Blank values are dropped.
`source`	A friendly label for the service.
`timeout`	Seconds before the request is abandoned.
`max_tries`	Total attempts, including the first.
`headers`	A named list of headers, all marked sensitive.
`throttle`	An optional throttle spec. See `req_defaults()`.
`secret_query`	A named list of query-string credentials, for a service that has no header form. Deliberately **not** part of the cache key, so it suits a credential that raises a rate limit and not one that changes the response. See `redact_secrets()`.

Details

The success-only cache means the file is fetched once per URL and every later lookup against it is served in process.

Value

An envelope whose `data` is a single string on success.

Examples

```
cache_reset()
breaker_reset()

httr2::with_mocked_responses(
  list(httr2::response(
    status_code = 200,
    body = charToRaw("gene\tclassification\nBRCA1\tDefinitive\n")
  )),
  get_text(
    "https://search.clinicalgenome.org",
    path = "kb/gene-validity/download",
    source = "ClinGen"
  )$data
)
```

`graphql_error` *Catch a GraphQL query error inside a 200*

Description

GraphQL reports query errors in the body of an HTTP 200, as a top-level `errors` array, so a 2xx status is not enough to call a call successful.

Usage

```
graphql_error(res, source = "API")
```

Arguments

`res`	An envelope from `post_json()` or `perform()`.
`source`	A friendly label for the service.

Details

Returns an error envelope for either kind of failure, transport or query, so a GraphQL client collapses the two checks into one:

```
res <- post_json(url, body, source = "gnomAD")
bad <- graphql_error(res, "gnomAD")
if (!is.null(bad)) return(bad)
```

Value

The failing envelope, or `NULL` when the call genuinely succeeded.

Examples

```
clean <- status_ok(data = list(data = list(gene = "BRCA1")), source = "G")
graphql_error(clean, "G")

queried <- status_ok(data = list(errors = list(list(message = "bad"))))
graphql_error(queried, "G")$status
```

`http_error_message` *A user-facing sentence for a failed call*

Description

Works out which status the failure is and asks `status_message()` for the wording, so it honours `biohttp.status_message` like everything else. Pass `http` for a response that arrived, or `condition` for a transport failure where none did.

Usage

```
http_error_message(source, http = NA_integer_, condition = NULL)
```

Arguments

`source`	A friendly label for the service, for example `"gnomAD"`.
`http`	An HTTP status code, or `NA_integer_` when no response arrived.
`condition`	A caught condition, used only when `http` is `NA`.

Value

A single string.

Examples

```
http_error_message("gnomAD", http = 404L)
http_error_message("gnomAD", http = 503L)
http_error_message("gnomAD", condition = simpleError("Timeout was reached"))
```

is_blank *Is a value blank*

Description

TRUE for NULL, a zero-length value, NA, or a string that is empty once trimmed. Used to drop empty query parameters so a request never sends param= with nothing after it.

Usage

```
is_blank(x)
```

Arguments

x	Any value.

Value

A single logical.

Examples

```
is_blank(NULL)
is_blank("  ")
is_blank("gnomAD")
```

is_transient *Is a response worth retrying*

Description

A 429 and the standard 5xx codes are transient. Everything else is a settled answer, and retrying it just spends someone else's capacity.

Usage

```
is_transient(resp)
```

Arguments

resp	An httr2 response.

Value

A single logical.

Examples

```
is_transient(httr2::response(status_code = 503))
is_transient(httr2::response(status_code = 404))
```

---

perform *Perform a request and normalize the result*

Description

Never raises on an HTTP or a parse failure. Returns an envelope for every outcome, so a caller branches on `res$status` and writes no `tryCatch()` of its own.

Usage

```
perform(req, source = "API", secret_query = NULL)
```

Arguments

`req`	An httr2 request, prepared with `req_defaults()`.
`source`	A friendly label for the service, used in the user-facing message. For example `"gnomAD"`.
`secret_query`	A named list of query-string credentials. Appended here rather than by the caller, so `req` itself never carries them and neither does anything built from `req$url`. See `redact_secrets()`.

Details

Short-circuits to a `skipped` envelope without sending anything when the host's breaker is open. See `breaker_open()`.

The request must have been through `req_defaults()`, which disarms httr2's error raising. Without that, `req_perform()` throws on a non-2xx and a perfectly reachable host is recorded as a transport failure.

Value

An envelope. See `envelope()`.

Examples

```
breaker_reset()
req <- req_defaults(httr2::request("https://mock.test/x"))

httr2::with_mocked_responses(
  list(httr2::response(
    status_code = 200,
    headers = list(`content-type` = "application/json"),
```

```
    body = charToRaw('{"symbol":"BRCA1"}')
  )),
  perform(req, "MyGene")$data$symbol
)
```

`perform_many` *Perform many requests as one batch*

Description

For many questions to one source. Returns a list of envelopes the same length as `reqs` and **in the same order**, so a caller zips the results back onto whatever it asked about by position.

Usage

```
perform_many(
  reqs,
  source = "API",
  max_active = 6,
  progress = FALSE,
  secret_query = NULL
)
```

Arguments

`reqs`	A list of httr2 requests, each prepared with `req_defaults()`.
`source`	A friendly label for the service, used in the user-facing message. For example `"MyGene"`.
`max_active`	Maximum requests in flight at once.
`progress`	Passed to `httr2::req_perform_parallel()`. `FALSE` by default, because the usual caller is a Shiny app that renders its own.
`secret_query`	A named list of query-string credentials, applied to every request in the batch at dispatch. See `redact_secrets()`.

Details

Each request must already have been through `req_defaults()`, the same as for `perform()`.

Value

A list of envelopes, the same length and order as `reqs`. See `envelope()`.

Same host, not a mix

httr2 applies `req_throttle()` and `req_retry()` across the whole list rather than per request, so a throttled request to one host makes an unthrottled request to another wait behind it. Requests are therefore grouped by host and each group is dispatched separately, which keeps each host's throttle bucket honest but means host groups run one after another.

Pass requests for one host. To query many different services at once, use process-level concurrency in the application instead.

Supply a throttle

httr2's own advice is never to perform in parallel without `req_throttle()`, because it is otherwise very easy to flood a source with simultaneous requests. `req_defaults()` takes a `throttle` argument and defaults its realm to the request's host. Use it. Public biological data sources are typically run on a research budget.

The breaker acts between batches, not inside one

A host with an open breaker is never dispatched to, and every one of its requests comes back `skipped`. But once a batch is in flight, a failure on the first request cannot short-circuit the rest, because they have already been sent. Transport failures within a batch are recorded, so they take effect on the next call rather than the current one.

Examples

```
breaker_reset()
reqs <- lapply(
  c("BRCA1", "TP53"),
  function(symbol) {
    req_defaults(httr2::request(paste0("https://mock.test/gene/", symbol)))
  }
)

httr2::with_mocked_responses(
  function(req) {
    httr2::response(
      status_code = 200,
      headers = list(`content-type` = "application/json"),
      body = charToRaw('{"ok":true}')
    )
  },
  vapply(perform_many(reqs, "MyGene"), function(res) res$status, character(1))
)
```

perform_text *Perform a request and return the body as text*

Description

The non-JSON counterpart to perform(), for a source that ships a bulk flat file rather than a per-record JSON API. The body lands in data as a single string, so the envelope shape is the same either way and a caller does not branch on which wrapper it used.

Usage

```
perform_text(req, source = "API", secret_query = NULL)
```

Arguments

req	An httr2 request, prepared with req_defaults().
source	A friendly label for the service, used in the user-facing message. For example "gnomAD".
secret_query	A named list of query-string credentials. Appended here rather than by the caller, so req itself never carries them and neither does anything built from req$url. See redact_secrets().

Details

The breaker rule is the same one: only a transport failure counts against the host.

Value

An envelope whose data is a single string on success.

Examples

```
breaker_reset()
req <- req_defaults(httr2::request("https://mock.test/genes.tsv"))

httr2::with_mocked_responses(
  list(httr2::response(
    status_code = 200,
    body = charToRaw("gene\tscore\n")
  )),
  perform_text(req, "ClinGen")$data
)
```

pluck_at *Pull a value out of a nested list*

Description

Walks a key path and returns `default` if any level is missing or `NULL`, rather than erroring. Parsed JSON is deeply nested and frequently missing branches, so this is the difference between a client that reads cleanly and one buried in `if (!is.null(...))`.

Usage

```
pluck_at(x, ..., default = NULL)
```

Arguments

`x`	A list, usually a parsed JSON body.
`...`	Keys to follow, outermost first.
`default`	Returned when any level of the path is missing or `NULL`.

Value

The value at the end of the path, or `default`.

Examples

```
body <- list(data = list(gene = list(symbol = "BRCA1")))
pluck_at(body, "data", "gene", "symbol")
pluck_at(body, "data", "variant", "id", default = NA_character_)
```

post_json *POST a JSON body*

Description

For a JSON or GraphQL POST. Caches on the URL and the body together, so two different queries to the same endpoint do not collide.

Usage

```
post_json(
  url,
  body,
  source = "API",
  timeout = 20,
  max_tries = 3,
  headers = NULL,
  throttle = NULL,
  secret_query = NULL
)
```

Arguments

url	The endpoint URL.
body	A list, serialized as the JSON request body.
source	A friendly label for the service.
timeout	Seconds before the request is abandoned.
max_tries	Total attempts, including the first.
headers	A named list of headers, all marked sensitive.
throttle	An optional throttle spec. See req_defaults().
secret_query	A named list of query-string credentials, for a service that has no header form. Deliberately **not** part of the cache key, so it suits a credential that raises a rate limit and not one that changes the response. See redact_secrets().

Value

An envelope. See envelope().

Examples

```
cache_reset()
breaker_reset()

httr2::with_mocked_responses(
  list(httr2::response(
    status_code = 200,
    headers = list(`content-type` = "application/json"),
    body = charToRaw('{"data":{"gene":{"gene_id":"ENSG00000012048"}}}')
  )),
  post_json(
    "https://gnomad.broadinstitute.org/api",
    body = list(query = "{ gene(gene_symbol: \"BRCA1\") { gene_id } }"),
    source = "gnomAD"
  )$data$data$gene$gene_id
)
```

post_json_many *POST many JSON bodies as one batch*

Description

The batched counterpart to post_json(), for a JSON or GraphQL endpoint answering many queries. Keyed on the URL and the body together, so two different queries to the same endpoint do not collide.

Usage

```
post_json_many(
  url,
  bodies = list(),
  source = "API",
  timeout = 20,
  max_tries = 3,
  headers = NULL,
  throttle = NULL,
  max_active = 6,
  progress = FALSE,
  secret_query = NULL
)
```

Arguments

url	The endpoint URL.
bodies	A list of request bodies, one per request. Each is serialized as JSON.
source	A friendly label for the service, used in the user-facing message. For example "MyGene".
timeout	Seconds before a request is abandoned.
max_tries	Total attempts, including the first. Note that httr2 does not honor this under parallel performance.
headers	A named list of headers, all marked sensitive.
throttle	A throttle spec. See req_defaults().
max_active	Maximum requests in flight at once.
progress	Passed to httr2::req_perform_parallel(). FALSE by default, because the usual caller is a Shiny app that renders its own.
secret_query	A named list of query-string credentials, applied to every request in the batch at dispatch. See redact_secrets().

Details

Read the sections on perform_many() first.

Value

A list of envelopes, the same length and order as `bodies`.

Examples

```
cache_reset()
breaker_reset()

httr2::with_mocked_responses(
  function(req) {
    httr2::response(
      status_code = 200,
      headers = list(`content-type` = "application/json"),
      body = charToRaw('{"data":{}}')
    )
  },
  length(post_json_many(
    "https://gnomad.broadinstitute.org/api",
    bodies = list(list(query = "{ a }"), list(query = "{ b }")),
    source = "gnomAD"
  ))
)
```

---

redact_secrets *Remove secret values from a string*

Description

Replaces each secret wherever it appears, in both the form it was passed and the percent-encoded form a URL carries. Used on the messages built from a transport failure, because a curl error normally carries the URL that failed and that URL may hold a query-string credential.

Usage

```
redact_secrets(text, secrets = NULL)
```

Arguments

`text`	A single string.
`secrets`	A named list of secret values, as passed to `secret_query`.

Details

Matching on the value rather than on the parameter name is deliberate: the same secret can reach a message through a URL, a header dump, or a proxy error, and only the value is common to all three.

Both forms have to be matched because a secret never reaches a URL in the form it was passed. `httr2::req_url_query()` percent-encodes it, so a key holding `+`, `/`, or `=`, which is an ordinary shape for a base64 key, arrives as `%2B`, `%2F`, and `%3D`. Matching only the raw value would let exactly those keys through into a logged `detail`.

Value

`text`, with every secret value replaced by `<redacted>`.

Examples

```
redact_secrets(
  "Could not resolve host: example.org/?api_key=abc123",
  list(api_key = "abc123")
)

# The encoded form is caught too, which is the one a URL actually carries.
redact_secrets(
  "Could not resolve host: example.org/?api_key=ab%2Bcd",
  list(api_key = "ab+cd")
)
```

---

req_defaults *Apply the shared request options*

Description

Timeout, bounded retry on transient failures, an attributable user agent, and optionally a throttle and headers.

Usage

```
req_defaults(
  req,
  timeout = 20,
  max_tries = 3,
  headers = NULL,
  throttle = NULL,
  user_agent = default_user_agent()
)
```

Arguments

`req`	An httr2 request.
`timeout`	Seconds before the request is abandoned.
`max_tries`	Total attempts, including the first.
`headers`	A named list of headers. All of them are marked sensitive.

throttle	An optional list with capacity, fill_time_s, and realm. httr2 token buckets are per process, so a realm keyed on the host gives one honest bucket per host. Defaults to the request's own host.
user_agent	The User-Agent string. See user_agent().

Details

req_error() is disarmed here on purpose. httr2 would otherwise raise on a non-2xx, and this package normalizes every outcome into an envelope instead, so perform() needs the response object rather than a condition.

Headers are attached with .redact set to their names, so a token never prints in an inspected request, a log line, or an error message. Header auth also keeps the secret out of the URL, where it would end up in access logs.

Value

The request, with the options applied.

Examples

```
req_defaults(httr2::request("https://example.org"), timeout = 5)

# A key-gated source. The token is redacted from anything printable.
req_defaults(
  httr2::request("https://example.org"),
  headers = list(Authorization = "Bearer secret-token")
)
```

---

status_constructors *Envelope constructors, one per status*

Description

Shorthands for envelope() that fill in the user-facing error sentence for their status. error is what a user reads; detail is what a log records.

Usage

```
status_ok(data, source = "API", http = 200L)

status_no_data(source = "API", http = NA_integer_, detail = NULL)

status_stale(data, source = "API", detail = NULL)

status_rate_limited(source = "API", http = 429L, detail = NULL)

status_timeout(source = "API", detail = NULL)
```

```
status_skipped(source = "API", detail = NULL)

status_error(source = "API", http = NA_integer_, error = NULL, detail = NULL)
```

Arguments

data	The parsed body on success, NULL otherwise.
source	A friendly label for the service, used in the user-facing message. For example "gnomAD".
http	The HTTP status code, or NA_integer_ when no response arrived.
detail	The technical cause, for a log. Never shown to a user.
error	One sentence fit to show a user. Never carries technical detail.

Details

The sentence comes from status_message(), so setting biohttp.status_message changes what these produce too.

Value

An envelope list, as described in envelope().

Examples

```
status_ok(data = list(n = 1), source = "MyGene")
status_no_data(source = "gnomAD", http = 404L)
status_skipped(source = "gnomAD", detail = "gnomad.broadinstitute.org open")
```

STATUS_LEVELS *The seven envelope statuses*

Description

Ordered from best outcome to worst.

Usage

```
STATUS_LEVELS
```

Format

A character vector of length seven.

Details

`ok` The call succeeded and `data` holds the parsed result.

`no_data` The source was reached and has nothing for this query, for example a 404 or an empty body. This is an answer, not a fault.

`stale` A cached value served past its freshness window.

`rate_limited` The source asked the caller to slow down, a 429.

`timeout` The request did not return in time.

`skipped` Not attempted, because the host's breaker is open or a caller-side budget was hit.

`error` Anything else: a 5xx, a malformed body, a transport failure.

`status_message` *The user-facing sentence for an outcome*

Description

Every `error` field in the package is produced here, so the seven statuses cannot drift apart and an application can replace all of them at once.

Usage

```
status_message(
  source = "API",
  status = "error",
  http = NA_integer_,
  condition = NULL
)
```

Arguments

`source`	A friendly label for the service, for example `"gnomAD"`.
`status`	One of STATUS_LEVELS.
`http`	An HTTP status code, or `NA_integer_` when no response arrived.
`condition`	A caught condition, used when no response arrived.

Value

A single string, or `NULL` for a status that carries no message.

Supplying your own wording

Set `biohttp.status_message` to a function of `source`, `status`, `http` and `condition`. It is called instead of the built-in, and it is the way to keep an app's own voice, or to localize, while adopting the transport:

```
options(biohttp.status_message = function(source, status, http, condition) {
  if (identical(status, "timeout")) {
    paste0(source, " took too long. Please try again shortly.")
  } else {
    NULL  # fall through to the built-in for everything else
  }
})
```

Returning anything other than a single non-empty string falls back to the built-in, so covering one case and leaving the rest is expected rather than an error.

An override cannot break the return-a-value contract. It runs inside a `tryCatch()`, and a function that raises is treated as though it returned nothing. This matters because the override runs on the failure path, which is precisely where the package promises never to raise.

Examples

```
status_message("gnomAD", "no_data")
status_message("gnomAD", "error", http = 503L)

old <- options(
  biohttp.status_message = function(source, status, http, condition) {
    paste0(source, " says: ", status)
  }
)
status_message("gnomAD", "timeout")
options(old)
```

---

user_agent *Build an attributable User-Agent string*

Description

Public sources ask callers to identify themselves. A caller identity plus a contact route let a source operator reach you before they rate-limit you.

Usage

```
user_agent(
  identity = env_chr("BIOHTTP_CALLER_IDENTITY", "biohttp"),
  version = as.character(getNamespaceVersion("biohttp")),
  url = env_chr("BIOHTTP_CONTACT_URL", ""),
  email = env_chr("BIOHTTP_CONTACT_EMAIL", "")
)
```

Arguments

identity	The calling application's name. Defaults to BIOHTTP_CALLER_IDENTITY, then to "biohttp".
version	The calling application's version.
url	A URL an operator can look up, usually the app's repository. Defaults to BIOHTTP_CONTACT_URL.
email	A contact address. Defaults to BIOHTTP_CONTACT_EMAIL. Left out of the string entirely when blank, rather than printed empty.

Details

The package deliberately does not ship an identity of its own. A consumer passes its own, or sets BIOHTTP_CALLER_IDENTITY, BIOHTTP_CONTACT_URL, and BIOHTTP_CONTACT_EMAIL so a hosted deployment can set them without touching code.

Value

A single string.

Examples

```
user_agent("my-shiny-app", "1.2.0", email = "ops@example.org")
user_agent("my-client-package", "0.4.0")
```

Index

∗ **datasets**
    STATUS_LEVELS, 29

as_legacy_envelope, 2

body_or_null, 3
body_or_null(), *11*
breaker_open, 4
breaker_open(), *19*
breaker_record, 5
breaker_reset, 5

cache, 6
cache(), *9*
cache_key, 8
cache_key(), *7*
cache_reset, 9
cache_reset(), *6*
cached, 7
cached(), *4*, *6*
classify_condition, 9
classify_http, 10

envelope, 11
envelope(), *2, 3*, *12*, *19, 20*, *24*, *28, 29*

get_json, 12
get_json(), *13*, *15*
get_json_many, 13
get_text, 15
graphql_error, 16

http_error_message, 17

is_blank, 18
is_transient, 18

perform, 19
perform(), *3, 4*, *16*, *20*, *22*, *28*
perform_many, 20
perform_many(), *14*, *25*
perform_text, 22
pluck_at, 23
post_json, 23
post_json(), *16*, *25*
post_json_many, 25

redact_secrets, 26
redact_secrets(), *12*, *14, 15*, *19, 20*, *22*, *24, 25*
req_defaults, 27
req_defaults(), *12*, *14, 15*, *19, 20*, *22*, *24, 25*

status_constructors, 28
status_error *(*status_constructors*)*, 28
STATUS_LEVELS, *10, 11*, 29, *30*
status_message, 30
status_message(), *17*, *29*
status_no_data *(*status_constructors*)*, 28
status_ok *(*status_constructors*)*, 28
status_rate_limited *(*status_constructors*)*, 28
status_skipped *(*status_constructors*)*, 28
status_stale *(*status_constructors*)*, 28
status_timeout *(*status_constructors*)*, 28

user_agent, 31
user_agent(), *28*