# Package 'gargle'

October 28, 2026

**Title** Utilities for Working with Google APIs

**Version** 1.7.0

**Description** Provides utilities for working with Google APIs <https://developers.google.com/apis-explorer>. This includes functions and classes for handling common credential types and for preparing, executing, and processing HTTP requests.

**License** MIT + file LICENSE

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

**BugReports** https://github.com/r-lib/gargle/issues

**Depends** R (>= 4.2)

**Imports** cli (>= 3.0.1), fs (>= 1.3.1), glue (>= 1.3.0), httr (>= 1.4.5), jsonlite, lifecycle, openssl, R6, rappdirs, rlang (>= 1.1.0), stats, utils, withr

**Suggests** aws.ec2metadata, aws.signature, covr, httpuv, knitr, rmarkdown, spelling, testthat (>= 3.1.7)

**VignetteBuilder** knitr

**Config/Needs/website** tidyverse/tidytemplate

**Config/roxygen2/version** 8.1.0

**Config/testthat/edition** 3

**Config/usethis/last-upkeep** 2026-10-20

**Encoding** UTF-8

**Language** en-US

**NeedsCompilation** no

**Author** Jennifer Bryan [aut, cre] (ORCID: <https://orcid.org/0000-0002-6983-2759>),
Craig Citro [aut],
Hadley Wickham [aut] (ORCID: <https://orcid.org/0000-0003-4757-117X>),
Google Inc [cph],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

**Maintainer** Jennifer Bryan <jenny@posit.co>

**Repository** CRAN

**Date/Publication** 2026-10-28 19:50:02 UTC

# Contents

AuthState-class . . . 2
credentials_app_default . . . 5
credentials_byo_oauth2 . . . 7
credentials_external_account . . . 8
credentials_gce . . . 10
credentials_service_account . . . 12
credentials_user_oauth2 . . . 13
cred_funs . . . 15
field_mask . . . 18
gargle2.0_token . . . 19
gargle_oauth_client_from_json . . . 20
gargle_oauth_sitrep . . . 22
gargle_options . . . 23
gargle_secret . . . 25
gce_instance_service_accounts . . . 27
init_AuthState . . . 28
request_develop . . . 29
request_make . . . 32
request_retry . . . 33
response_process . . . 36
token-info . . . 38
token_fetch . . . 39

**Index** **41**

---

`AuthState-class` *Authorization state*

---

## Description

An `AuthState` object manages an authorization state, typically on behalf of a wrapper package that makes requests to a Google API.

The `vignette("gargle-auth-in-client-package")` describes a design for wrapper packages that relies on an `AuthState` object. This state can then be incorporated into the package's requests for tokens and can control the inclusion of tokens in requests to the target API.

- `api_key` is the simplest way to associate a request with a specific Google Cloud Platform project. A few calls to certain APIs, e.g. reading a public Sheet, can succeed with an API key, but this is the exception.
- `client` is an OAuth client ID (and secret) associated with a specific Google Cloud Platform project. This is used in the OAuth flow, in which an authenticated user authorizes the client to access or manipulate data on their behalf.

- `auth_active` reflects whether outgoing requests will be authorized by an authenticated user or are unauthorized requests for public resources. These two states correspond to sending a request with a token versus an API key, respectively.
- `cred` is where the current token is cached within a session, once one has been fetched. It is generally assumed to be an instance of `httr::TokenServiceAccount` or `httr::Token2.0` (or a subclass thereof), probably obtained via `token_fetch()` (or one of its constituent credential fetching functions).

An `AuthState` should be created through the constructor function `init_AuthState()`, which has more details on the arguments.

## Public fields

`package` Package name.

`client` An OAuth client.

`app` **[Deprecated]** Use `client` instead.

`api_key` An API key.

`auth_active` Logical, indicating whether auth is active.

`cred` Credentials.

## Methods

### Public methods:

- `AuthState$new()`
- `AuthState$format()`
- `AuthState$set_client()`
- `AuthState$set_app()`
- `AuthState$set_api_key()`
- `AuthState$set_auth_active()`
- `AuthState$set_cred()`
- `AuthState$clear_cred()`
- `AuthState$get_cred()`
- `AuthState$has_cred()`
- `AuthState$clone()`

`AuthState$new()`**:** Create a new AuthState

*Usage:*

```
AuthState$new(
  package = NA_character_,
  client = NULL,
  api_key = NULL,
  auth_active = TRUE,
  cred = NULL,
  app = deprecated()
)
```

*Arguments:*

`package` Package name.

`client` An OAuth client.

`api_key` An API key.

`auth_active` Logical, indicating whether auth is active.

`cred` Credentials.

`app` **[Deprecated]** Use `client` instead.

*Details:* For more details on the parameters, see `init_AuthState()`

`AuthState$format()`**:** Format an AuthState

*Usage:*

```
AuthState$format(...)
```

*Arguments:*

`...` Not used.

`AuthState$set_client()`**:** Set the OAuth client

*Usage:*

```
AuthState$set_client(client)
```

*Arguments:*

`client` An OAuth client.

`AuthState$set_app()`**: [Deprecated]** Deprecated method to set the OAuth client

*Usage:*

```
AuthState$set_app(app)
```

*Arguments:*

`app` **[Deprecated]** Use `client` instead.

`AuthState$set_api_key()`**:** Set the API key

*Usage:*

```
AuthState$set_api_key(value)
```

*Arguments:*

`value` An API key.

`AuthState$set_auth_active()`**:** Set whether auth is (in)active

*Usage:*

```
AuthState$set_auth_active(value)
```

*Arguments:*

`value` Logical, indicating whether to send requests authorized with user credentials.

`AuthState$set_cred()`**:** Set credentials

*Usage:*

```
AuthState$set_cred(cred)
```

*Arguments:*

`cred` User credentials.

`AuthState$clear_cred()`: Clear credentials

*Usage:*

```
AuthState$clear_cred()
```

`AuthState$get_cred()`: Get credentials

*Usage:*

```
AuthState$get_cred()
```

`AuthState$has_cred()`: Report if we have credentials

*Usage:*

```
AuthState$has_cred()
```

`AuthState$clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
AuthState$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

---

`credentials_app_default` *Load Application Default Credentials*

---

## Description

Loads credentials from a file identified via a search strategy known as Application Default Credentials (ADC). The hope is to make auth "just work" for someone working on Google-provided infrastructure or who has used Google tooling to get started, such as the gcloud command line tool.

A sequence of paths is consulted, which we describe here, with some abuse of notation. ALL_CAPS represents the value of an environment variable and `%||%` is used in the spirit of a null coalescing operator.

```
GOOGLE_APPLICATION_CREDENTIALS
CLOUDSDK_CONFIG/application_default_credentials.json
# on Windows:
(APPDATA %||% SystemDrive %||% C:)\gcloud\application_default_credentials.json
# on not-Windows:
~/.config/gcloud/application_default_credentials.json
```

If the above search successfully identifies a JSON file, it is parsed and ingested as a service account, an external account ("workload identity federation"), or a user account. Literally, if the JSON describes a service account, we call `credentials_service_account()` and if it describes an external account, we call `credentials_external_account()`.

## Usage

```
credentials_app_default(
  scopes = "https://www.googleapis.com/auth/cloud-platform",
  ...,
  subject = NULL
)
```

## Arguments

| | |
|---|---|
| scopes | A character vector of scopes to request. Pick from those listed at https://developers.google.com/identity/protocols/oauth2/scopes.<br>For certain token flows, the "https://www.googleapis.com/auth/userinfo.email" scope is unconditionally included. This grants permission to retrieve the email address associated with a token; gargle uses this to index cached OAuth tokens. This grants no permission to view or send email and is generally considered a low-value scope. |
| ... | Additional arguments passed to all credential functions. |
| subject | An optional subject claim. Specify this if you wish to use the service account represented by path to impersonate the subject, who is a normal user. Before this can work, an administrator must grant the service account domain-wide authority. Identify the user to impersonate via their email, e.g. subject = "user@example.com". Note that gargle automatically adds the non-sensitive "https://www.googleapis.com/auth/userinfo.email" scope, so this scope must be enabled for the service account, along with any other scopes being requested. |

## Value

An httr::TokenServiceAccount, a WifToken, an httr::Token2.0 or NULL.

## See Also

- https://docs.cloud.google.com/docs/authentication
- https://docs.cloud.google.com/sdk/docs

Other credential functions: credentials_byo_oauth2(), credentials_external_account(), credentials_gce(), credentials_service_account(), credentials_user_oauth2(), token_fetch()

## Examples

```
## Not run:
credentials_app_default()

## End(Not run)
```

credentials_byo_oauth2 *Load a user-provided token*

## Description

This function is designed to pass its `token` input through, after doing a few checks and some light processing:

- If `token` has class `request`, i.e. it is a token that has been prepared with `httr::config()`, the `auth_token` component is extracted. For example, such input could be returned by `googledrive::drive_token()` or `bigrquery::bq_token()`.
- If `token` is an instance of `Gargle2.0` (so: a gargle-obtained user token), checks that it appears to be a Google OAuth token, based on its embedded `oauth_endpoint`. Refreshes the token, if it's refreshable.
- Returns the `token`.

There is no point in providing `scopes`. They are ignored because the `scopes` associated with the token have already been baked in to the token itself and gargle does not support incremental authorization. The main point of `credentials_byo_oauth2()` is to allow `token_fetch()` (and packages that wrap it) to accommodate a "bring your own token" workflow.

This also makes it possible to obtain a token with one package and then register it for use with another package. For example, the default scope requested by googledrive is also sufficient for operations available in googlesheets4. You could use a shared token like so:

```
library(googledrive)
library(googlesheets4)
drive_auth(email = "jane_doe@example.com")
gs4_auth(token = drive_token())
# work with both packages freely now, with the same identity
```

## Usage

```
credentials_byo_oauth2(scopes = NULL, token, ...)
```

## Arguments

| | |
|---|---|
| `scopes` | A character vector of scopes to request. Pick from those listed at https://developers.google.com/identity/protocols/oauth2/scopes.<br>For certain token flows, the `"https://www.googleapis.com/auth/userinfo.email"` scope is unconditionally included. This grants permission to retrieve the email address associated with a token; gargle uses this to index cached OAuth tokens. This grants no permission to view or send email and is generally considered a low-value scope. |
| `token` | A token with class Token2.0 or an object of httr's class `request`, i.e. a token that has been prepared with `httr::config()` and has a Token2.0 in the `auth_token` component. |
| `...` | Additional arguments passed to all credential functions. |

## Value

An Token2.0.

## See Also

Other credential functions: `credentials_app_default()`, `credentials_external_account()`, `credentials_gce()`, `credentials_service_account()`, `credentials_user_oauth2()`, `token_fetch()`

## Examples

```
## Not run:
# assume `my_token` is a Token2.0 object returned by a function such as
# credentials_user_oauth2()
credentials_byo_oauth2(token = my_token)

## End(Not run)
```

---

# credentials_external_account

*Get a token for an external account*

---

## Description

**[Experimental]** Workload identity federation lets applications running outside Google Cloud, such as on AWS or GitHub Actions, access Google Cloud resources by using a federated identity instead of a service account key. This eliminates the dilemma of how to safely manage service account credential files.

Unlike a service account key, the configuration for workload identity federation is not a long-lived secret that you need to store and protect. What the configuration looks like depends on the platform.

- On AWS, it's a file of non-sensitive metadata that you download once and keep on the instance; the sensitive data is obtained "on-the-fly" at run time.
- On GitHub Actions, the configuration file is written fresh for each job and contains only a short-lived credential.

The configuration is used to obtain a so-called subject token from the external identity provider, such as AWS or GitHub. The subject token is then sent to Google's Security Token Service API, in exchange for a very short-lived federated access token. Finally, the federated access token is sent to Google's Service Account Credentials API, in exchange for a short-lived GCP access token. This access token allows the external application to impersonate a service account and inherit the permissions of the service account to access GCP resources.

This feature is still experimental in gargle. The `credential_source` in the configuration determines how the subject token is obtained and gargle supports these types:

- AWS (`"environment_id": "aws1"`). This only works when running on an EC2 instance and requires the suggested packages **aws.signature** and **aws.ec2metadata**.

- URL-sourced credentials (`"url"`). The subject token is fetched from a URL, with optional request `headers`. The `google-github-actions/auth` action writes this type of configuration for GitHub Actions and points the `GOOGLE_APPLICATION_CREDENTIALS` environment variable at it. That means that, in a workflow that runs this action, `token_fetch()` (and, therefore, the auth functions of client packages) can discover and use the credentials via `credentials_app_default()`.
- File-sourced credentials (`"file"`). The subject token is read from a local file.

For URL- and file-sourced credentials, the subject token is either the entire response or file (`"format": {"type": "text"}`, the default) or a field in a JSON object (`"format": {"type": "json", "subject_token_field_name"`

gargle does not support executable-sourced credentials. The configuration must include `service_account_impersonation_` i.e. gargle does not yet support direct workload identity federation, without service account impersonation. If you would like gargle to support additional variations of this token flow, please open an issue on GitHub and describe your use case.

## Usage

```
credentials_external_account(
  scopes = "https://www.googleapis.com/auth/cloud-platform",
  path = "",
  ...
)
```

## Arguments

| | |
|---|---|
| `scopes` | A character vector of scopes to request. Pick from those listed at `https://developers.google.com/identity/protocols/oauth2/scopes`.<br>For certain token flows, the `"https://www.googleapis.com/auth/userinfo.email"` scope is unconditionally included. This grants permission to retrieve the email address associated with a token; gargle uses this to index cached OAuth tokens. This grants no permission to view or send email and is generally considered a low-value scope. |
| `path` | JSON containing the workload identity configuration for the external account, in one of the forms supported for the `txt` argument of `jsonlite::fromJSON()` (probably, a file path, although it could be a JSON string). The instructions for generating this configuration are given at Configuring workload identity federation.<br>Note that external account tokens are a natural fit for use as Application Default Credentials, so consider storing the configuration file in one of the standard locations consulted for ADC, instead of providing `path` explicitly. See `credentials_app_default()` for more. |
| `...` | Additional arguments passed to all credential functions. |

## Value

A `WifToken()` or `NULL`.

## See Also

Most of the setup for this authentication method happens on the GCP side; the external platform typically needs little or no configuration. These links provide, respectively, a high-level overview, step-by-step instructions, and Google's specification for the configuration file.

- https://cloud.google.com/blog/products/identity-security/enable-keyless-access-to-gcp-with-wor
- https://cloud.google.com/iam/docs/configuring-workload-identity-federation
- https://google.aip.dev/auth/4117

Other credential functions: credentials_app_default(), credentials_byo_oauth2(), credentials_gce(), credentials_service_account(), credentials_user_oauth2(), token_fetch()

## Examples

```
## Not run:
credentials_external_account()

## End(Not run)
```

---

`credentials_gce` *Get a token from the Google metadata server*

---

## Description

If your code is running on Google Cloud, we can often obtain a token for an attached service account directly from a metadata server. This is more secure than working with an explicit a service account key, as credentials_service_account() does, and is the preferred method of auth for workloads running on Google Cloud.

The most straightforward scenario is when you are working in a VM on Google Compute Engine and it's OK to use the default service account. This should "just work" automatically.

`credentials_gce()` supports other use cases (such as GKE Workload Identity), but may require some explicit setup, such as:

- Create a service account, grant it appropriate scopes(s) and IAM roles, attach it to the target resource. This prep work happens outside of R, e.g., in the Google Cloud Console. On the R side, provide the email address of this appropriately configured service account via `service_account`.
- Specify details for constructing the root URL of the metadata service:
  - The logical option `"gargle.gce.use_ip"`. If undefined, this defaults to `FALSE`.
  - The environment variable `GCE_METADATA_URL` is consulted when `"gargle.gce.use_ip"` is `FALSE`. If undefined, the default is `metadata.google.internal`.
  - The environment variable `GCE_METADATA_IP` is consulted when `"gargle.gce.use_ip"` is `TRUE`. If undefined, the default is `169.254.169.254`.

- Change (presumably increase) the timeout for requests to the metadata server via the "gargle.gce.timeout" global option. This timeout is given in seconds and is set to a value (strategy, really) that often works well in practice. However, in some cases it may be necessary to increase the timeout with code such as:

```
options(gargle.gce.timeout = 3)
```

For details on specific use cases, such as Google Kubernetes Engine (GKE), see vignette("non-interactive-auth").

## Usage

```
credentials_gce(
  scopes = "https://www.googleapis.com/auth/cloud-platform",
  service_account = "default",
  ...
)
```

## Arguments

| | |
|---|---|
| scopes | A character vector of scopes to request. Pick from those listed at https://developers.google.com/identity/protocols/oauth2/scopes.<br>For certain token flows, the "https://www.googleapis.com/auth/userinfo.email" scope is unconditionally included. This grants permission to retrieve the email address associated with a token; gargle uses this to index cached OAuth tokens. This grants no permission to view or send email and is generally considered a low-value scope. |
| service_account | Name of the GCE service account to use. |
| ... | Additional arguments passed to all credential functions. |

## Value

A GceToken() or NULL.

## See Also

A related auth flow that can be used on certain non-Google platforms, such as AWS or GitHub Actions, is workload identity federation, which is implemented in credentials_external_account().

https://docs.cloud.google.com/compute/docs/access/service-accounts

https://docs.cloud.google.com/iam/docs/best-practices-service-accounts

How to attach a service account to a resource: https://cloud.google.com/iam/docs/impersonating-service-account
attaching-to-resources

https://docs.cloud.google.com/kubernetes-engine/docs/concepts/workload-identity

https://docs.cloud.google.com/kubernetes-engine/docs/how-to/workload-identity

https://docs.cloud.google.com/compute/docs/metadata/overview

Other credential functions: credentials_app_default(), credentials_byo_oauth2(), credentials_external_accour
credentials_service_account(), credentials_user_oauth2(), token_fetch()

## Examples

```
## Not run:
credentials_gce()

## End(Not run)
```

---

`credentials_service_account` *Load a service account token*

---

## Description

Load a service account token

## Usage

```
credentials_service_account(scopes = NULL, path = "", ..., subject = NULL)
```

## Arguments

| | |
|---|---|
| `scopes` | A character vector of scopes to request. Pick from those listed at https://developers.google.com/identity/protocols/oauth2/scopes.<br>For certain token flows, the `"https://www.googleapis.com/auth/userinfo.email"` scope is unconditionally included. This grants permission to retrieve the email address associated with a token; gargle uses this to index cached OAuth tokens. This grants no permission to view or send email and is generally considered a low-value scope. |
| `path` | JSON identifying the service account, in one of the forms supported for the `txt` argument of `jsonlite::fromJSON()` (typically, a file path or JSON string). |
| `...` | Additional arguments passed to all credential functions. |
| `subject` | An optional subject claim. Specify this if you wish to use the service account represented by `path` to impersonate the `subject`, who is a normal user. Before this can work, an administrator must grant the service account domain-wide authority. Identify the user to impersonate via their email, e.g. `subject = "user@example.com"`. Note that gargle automatically adds the non-sensitive `"https://www.googleapis.com/auth/userinfo.email"` scope, so this scope must be enabled for the service account, along with any other `scopes` being requested. |

## Details

Note that fetching a token for a service account requires a reasonably accurate system clock. For more information, see the `vignette("how-gargle-gets-tokens")`.

## Value

An `httr::TokenServiceAccount` or `NULL`.

## See Also

Additional reading on delegation of domain-wide authority:

- https://developers.google.com/identity/protocols/oauth2/service-account#delegatingauthority

Other credential functions: credentials_app_default(), credentials_byo_oauth2(), credentials_external_accour credentials_gce(), credentials_user_oauth2(), token_fetch()

## Examples

```
## Not run:
token <- credentials_service_account(
  scopes = "https://www.googleapis.com/auth/userinfo.email",
  path = "/path/to/your/service-account.json"
)

## End(Not run)
```

---

# credentials_user_oauth2

*Get an OAuth token for a user*

---

## Description

Consults the token cache for a suitable OAuth token and, if unsuccessful, gets a token via the browser flow. A cached token is suitable if it's compatible with the user's request in this sense:

- OAuth client must be same.
- Scopes must be same.
- Email, if provided, must be same. If specified email is a glob pattern like `"*@example.com"`, email matching is done at the domain level.

gargle is very conservative about using OAuth tokens discovered in the user's cache and will generally seek interactive confirmation. Therefore, in a non-interactive setting, it's important to explicitly specify the `"email"` of the target account or to explicitly authorize automatic discovery. See gargle2.0_token(), which this function wraps, for more. Non-interactive use also suggests it might be time to use a service account token or workload identity federation.

## Usage

```
credentials_user_oauth2(
  scopes = NULL,
  client = gargle_client(),
  package = "gargle",
  ...,
  app = deprecated()
)
```

## Arguments

| | |
|---|---|
| scopes | A character vector of scopes to request. Pick from those listed at https://developers.google.com/identity/protocols/oauth2/scopes.<br>For certain token flows, the "https://www.googleapis.com/auth/userinfo.email" scope is unconditionally included. This grants permission to retrieve the email address associated with a token; gargle uses this to index cached OAuth tokens. This grants no permission to view or send email and is generally considered a low-value scope. |
| client | A Google OAuth client, preferably constructed via gargle_oauth_client_from_json(), which returns an instance of gargle_oauth_client. For backwards compatibility, for a limited time, gargle will still accept an "OAuth app" created with httr::oauth_app(). |
| package | Name of the package requesting a token. Used in messages. |
| ... | Arguments passed on to gargle2.0_token<br>email Optional. If specified, email can take several different forms:<br>• "jane@gmail.com", i.e. an actual email address. This allows the user to target a specific Google identity. If specified, this is used for token lookup, i.e. to determine if a suitable token is already available in the cache. If no such token is found, email is used to pre-select the targeted Google identity in the OAuth chooser. (Note, however, that the email associated with a token when it's cached is always determined from the token itself, never from this argument).<br>• "*@example.com", i.e. a domain-only glob pattern. This can be helpful if you need code that "just works" for both alice@example.com and bob@example.com.<br>• TRUE means that you are approving email auto-discovery. If exactly one matching token is found in the cache, it will be used.<br>• FALSE or NA mean that you want to ignore the token cache and force a new OAuth dance in the browser.<br>Defaults to the option named "gargle_oauth_email", retrieved by gargle_oauth_email() (unless a wrapper package implements different default behavior).<br>use_oob Whether to use out-of-band authentication (or, perhaps, a variant implemented by gargle and known as "pseudo-OOB") when first acquiring the token. Defaults to the value returned by gargle_oob_default(). Note that (pseudo-)OOB auth only affects the initial OAuth dance. If we retrieve (and possibly refresh) a cached token, use_oob has no effect.<br>If the OAuth client is provided implicitly by a wrapper package, its type probably defaults to the value returned by gargle_oauth_client_type(). You can take control of the client type by setting options(gargle_oauth_client_type = "web") or options(gargle_oauth_client_type = "installed").<br>cache Specifies the OAuth token cache. Defaults to the option named "gargle_oauth_cache", retrieved via gargle_oauth_cache().<br>credentials Advanced use only: allows you to completely customise token generation. |
| app | **[Deprecated]** Replaced by the client argument. |

## Value

A Gargle2.0 token.

## See Also

Other credential functions: credentials_app_default(), credentials_byo_oauth2(), credentials_external_accour
credentials_gce(), credentials_service_account(), token_fetch()

## Examples

```
## Not run:
# Drive scope, built-in gargle demo client
scopes <- "https://www.googleapis.com/auth/drive"
credentials_user_oauth2(scopes, client = gargle_client())

# bring your own client
client <- gargle_oauth_client_from_json(
  path = "/path/to/the/JSON/you/downloaded/from/gcp/console.json",
  name = "my-nifty-oauth-client"
)
credentials_user_oauth2(scopes, client)

## End(Not run)
```

---

# cred_funs *Credential function registry*

---

## Description

Functions to query or manipulate the registry of credential functions consulted by token_fetch().

## Usage

```
cred_funs_list()

cred_funs_add(...)

cred_funs_set(funs, ls = deprecated())

cred_funs_clear()

cred_funs_list_default()

cred_funs_set_default()

local_cred_funs(
  funs = cred_funs_list_default(),
  action = c("replace", "modify"),
```

```
  .local_envir = caller_env()
)

with_cred_funs(
  funs = cred_funs_list_default(),
  code,
  action = c("replace", "modify")
)
```

## Arguments

| | |
|---|---|
| `...` | <dynamic-dots> One or more credential functions, in `name = value` form. Each credential function is subject to a superficial check that it at least "smells like" a credential function: its first argument must be named `scopes`, and its signature must include `...`. To remove a credential function, you can use a specification like `name = NULL`. |
| `funs` | A named list of credential functions. |
| `ls` | **[Deprecated]** This argument has been renamed to `funs`. |
| `action` | Whether to use `funs` to replace or modify the registry with funs:<br>• `"replace"` does `cred_funs_set(funs)`<br>• `"modify"` does `cred_funs_add(!!!funs)` |
| `.local_envir` | The environment to use for scoping. Defaults to current execution environment. |
| `code` | Code to run with temporary credential function registry. |

## Value

A list of credential functions or `NULL`.

## Functions

- `cred_funs_list()`: Get the list of registered credential functions.
- `cred_funs_add()`: Register one or more new credential fetching functions. Function(s) are added to the *front* of the list. So:
  - "First registered, last tried."
  - "Last registered, first tried."

  Can also be used to *remove* a function from the registry.
- `cred_funs_set()`: Register a list of credential fetching functions.
- `cred_funs_clear()`: Clear the credential function registry.
- `cred_funs_list_default()`: Return the default list of credential functions.
- `cred_funs_set_default()`: Reset the registry to the gargle default.
- `local_cred_funs()`: Modify the credential function registry in the current scope. It is an example of the `local_*()` functions in **withr**.
- `with_cred_funs()`: Evaluate `code` with a temporarily modified credential function registry. It is an example of the `with_*()` functions in **withr**.

## See Also

token_fetch(), which is where the registry is actually used.

## Examples

```
names(cred_funs_list())

creds_one <- function(scopes, ...) {}

cred_funs_add(one = creds_one)
cred_funs_add(two = creds_one, three = creds_one)
names(cred_funs_list())

cred_funs_add(two = NULL)
names(cred_funs_list())

# restore the default list
cred_funs_set_default()

# remove one specific credential fetcher
cred_funs_add(credentials_gce = NULL)
names(cred_funs_list())

# force the use of one specific credential fetcher
cred_funs_set(list(credentials_user_oauth2 = credentials_user_oauth2))
names(cred_funs_list())

# restore the default list
cred_funs_set_default()

# run some code with a temporary change to the registry
# creds_one ONLY
with_cred_funs(
  list(one = creds_one),
  names(cred_funs_list())
)
# add creds_one to the list
with_cred_funs(
  list(one = creds_one),
  names(cred_funs_list()),
  action = "modify"
)
# remove credentials_gce
with_cred_funs(
  list(credentials_gce = NULL),
  names(cred_funs_list()),
  action = "modify"
)
```

`field_mask` *Generate a field mask*

## Description

Many Google API requests take a field mask, via a `fields` parameter, in the URL and/or in the body. `field_mask()` generates such a field mask from an R list, typically a list that is destined to be part of the body of a request that writes or updates a resource. `field_mask()` is designed to help in the common case where the attributes you wish to modify are exactly the ones represented in the object. It is possible to use a "larger" field mask, that is either less specific or that explicitly includes other attributes, in which case the attributes covered by the mask but absent from the object are reset to default values. This is not exactly the use case `field_mask()` is designed for, but its output could still be useful as a first step in constructing such a mask.

## Usage

```
field_mask(x)
```

## Arguments

| | |
|---|---|
| `x` | A named R list, where the requirement for names applies at all levels, i.e. recursively. |

## Value

A Google API field mask, as a string.

## See Also

The documentation for the JSON encoding of a Protocol Buffers FieldMask.

## Examples

```
x <- list(sheetId = 1234, title = "my_favorite_worksheet")
field_mask(x)

x <- list(
  userEnteredFormat = list(
    backgroundColor = list(
      red = 159 / 255, green = 183 / 255, blue = 196 / 255
    )
  )
)
field_mask(x)

x <- list(
  sheetId = 1234,
  gridProperties = list(rowCount = 5, columnCount = 3)
)
field_mask(x)
```

gargle2.0_token *Generate a gargle token*

## Description

Constructor function for objects of class Gargle2.0.

## Usage

```
gargle2.0_token(
  email = gargle_oauth_email(),
  client = gargle_client(),
  package = "gargle",
  scope = NULL,
  use_oob = gargle_oob_default(),
  credentials = NULL,
  cache = if (is.null(credentials)) gargle_oauth_cache() else FALSE,
  ...,
  app = deprecated()
)
```

## Arguments

| | |
|---|---|
| email | Optional. If specified, email can take several different forms:<br>• "jane@gmail.com", i.e. an actual email address. This allows the user to target a specific Google identity. If specified, this is used for token lookup, i.e. to determine if a suitable token is already available in the cache. If no such token is found, email is used to pre-select the targeted Google identity in the OAuth chooser. (Note, however, that the email associated with a token when it's cached is always determined from the token itself, never from this argument).<br>• "*@example.com", i.e. a domain-only glob pattern. This can be helpful if you need code that "just works" for both alice@example.com and bob@example.com.<br>• TRUE means that you are approving email auto-discovery. If exactly one matching token is found in the cache, it will be used.<br>• FALSE or NA mean that you want to ignore the token cache and force a new OAuth dance in the browser.<br>Defaults to the option named "gargle_oauth_email", retrieved by gargle_oauth_email() (unless a wrapper package implements different default behavior). |
| client | A Google OAuth client, preferably constructed via gargle_oauth_client_from_json(), which returns an instance of gargle_oauth_client. For backwards compatibility, for a limited time, gargle will still accept an "OAuth app" created with httr::oauth_app(). |
| package | Name of the package requesting a token. Used in messages. |

| | |
|---|---|
| scope | A character vector of scopes to request. |
| use_oob | Whether to use out-of-band authentication (or, perhaps, a variant implemented by gargle and known as "pseudo-OOB") when first acquiring the token. Defaults to the value returned by `gargle_oob_default()`. Note that (pseudo-)OOB auth only affects the initial OAuth dance. If we retrieve (and possibly refresh) a cached token, use_oob has no effect.<br>If the OAuth client is provided implicitly by a wrapper package, its type probably defaults to the value returned by `gargle_oauth_client_type()`. You can take control of the client type by setting `options(gargle_oauth_client_type = "web")` or `options(gargle_oauth_client_type = "installed")`. |
| credentials | Advanced use only: allows you to completely customise token generation. |
| cache | Specifies the OAuth token cache. Defaults to the option named `"gargle_oauth_cache"`, retrieved via `gargle_oauth_cache()`. |
| ... | Absorbs arguments intended for use by other credential functions. Not used. |
| app | **[Deprecated]** Replaced by the `client` argument. |

## Value

An object of class Gargle2.0, either new or loaded from the cache.

## Examples

```
## Not run:
gargle2.0_token()

## End(Not run)
```

---

# gargle_oauth_client_from_json

*Create an OAuth client for Google*

---

## Description

A `gargle_oauth_client` consists of:

- A type. gargle only supports the "Desktop app" and "Web application" client types. Different types are associated with different OAuth flows.
- A client ID and secret.
- Optionally, one or more redirect URIs.
- A name. This is really a human-facing label. Or, rather, it can be used that way, but the default is just a hash. We recommend using the same name here as the name used to label the client ID in the Google Cloud Platform Console.

A `gargle_oauth_client` is an adaptation of httr's `oauth_app()` (currently) and httr2's `oauth_client()` (which gargle will migrate to in the future).

## Usage

```
gargle_oauth_client_from_json(path, name = NULL)

gargle_oauth_client(
  id,
  secret,
  redirect_uris = NULL,
  type = c("installed", "web"),
  name = hash(id)
)
```

## Arguments

| | |
|---|---|
| path | JSON downloaded from Google Cloud Console, containing a client id and secret, in one of the forms supported for the `txt` argument of `jsonlite::fromJSON()` (typically, a file path or JSON string). |
| name | A label for this specific client, presumably the same name used to label it in Google Cloud Console. Unfortunately there is no way to make that true programmatically, i.e. the JSON representation does not contain this information. |
| id | Client ID |
| secret | Client secret |
| redirect_uris | Where your application listens for the response from Google's authorization server. If you didn't configure this specifically when creating the client (which is only possible for clients of the "web" type), you can leave this unspecified. |
| type | Specifies the type of OAuth client. The valid values are a subset of possible Google client types and reflect the key used to describe the client in its JSON representation:<br>• `"installed"` is associated with a "Desktop app"<br>• `"web"` is associated with a "Web application" |

## Value

An OAuth client: An S3 list with class `gargle_oauth_client`. For backwards compatibility reasons, this currently also inherits from the httr S3 class `oauth_app`, but that is a temporary measure. An instance of `gargle_oauth_client` stores more information than httr's `oauth_app`, such as the OAuth client's type ("web" or "installed").

There are some redundant fields in this object during the httr-to-httr2 transition period. The legacy fields `appname` and `key` repeat the information in the future-facing fields `name` and (client) `id`. Prefer `name` and `id` to `appname` and `key` in downstream code. Prefer the constructors `gargle_oauth_client_from_json()` and `gargle_oauth_client()` to `httr::oauth_app()` and `oauth_app_from_json()`.

## Examples

```
## Not run:
gargle_oauth_client_from_json(
  path = "/path/to/the/JSON/you/downloaded/from/gcp/console.json",
  name = "my-nifty-oauth-client"
```

```
)

## End(Not run)

gargle_oauth_client(
  id = "some_long_id",
  secret = "sssshhhhh_its_a_secret",
  name = "my-nifty-oauth-client"
)
```

---

`gargle_oauth_sitrep` *OAuth token situation report*

---

## Description

Get a human-oriented overview of the existing gargle OAuth tokens:

- Filepath of the current cache
- Number of tokens found there
- Compact summary of the associated
  - Email = Google identity
  - OAuth client (actually, just its nickname)
  - Scopes
  - Hash (actually, just the first 7 characters)

This is the most direct way to find out where gargle's OAuth cache is (see `gargle_oauth_cache()` for more about the cache location). The `filepath` column of the returned data frame makes it easy to delete specific tokens, e.g.:

```
dat <- gargle_oauth_sitrep()
# delete all cached tokens for one Google identity
unlink(dat$filepath[dat$email == "jane@example.com"])
```

## Usage

```
gargle_oauth_sitrep(cache = NULL)
```

## Arguments

| | |
|---|---|
| `cache` | Specifies the OAuth token cache. Defaults to the option named `"gargle_oauth_cache"`, retrieved via `gargle_oauth_cache()`. |

## Value

A data frame with one row per cached token, invisibly. Note this data frame contains more columns than it seems, e.g. the `filepath` column isn't printed by default.

## Examples

```
gargle_oauth_sitrep()
```

gargle_options *Options consulted by gargle*

## Description

Wrapper functions around options consulted by gargle, which provide:

- A place to hang documentation.
- The mechanism for setting a default.

If the built-in defaults don't suit you, set one or more of these options. Typically, this is done in the `.Rprofile` startup file, with code along these lines:

```
options(
  gargle_oauth_email = "jane@example.com",
  gargle_oauth_cache = "/path/to/folder/that/does/not/sync/to/cloud"
)
```

## Usage

```
gargle_oauth_email()

gargle_oob_default()

gargle_oauth_cache()

gargle_oauth_client_type()

gargle_verbosity()

local_gargle_verbosity(level, env = caller_env())

with_gargle_verbosity(level, code)
```

## Arguments

| | |
|---|---|
| `level` | Verbosity level: "debug" > "info" > "silent" |
| `env` | The environment to use for scoping |
| `code` | Code to execute with specified verbosity level |

`gargle_oauth_email`

`gargle_oauth_email()` returns the option named "gargle_oauth_email", which is undefined by default. If set, this option should be one of:

- An actual email address corresponding to your preferred Google identity. Example: `janedoe@gmail.com`.
- A glob pattern that indicates your preferred Google domain. Example: `*@example.com`.

- TRUE to allow email and OAuth token auto-discovery, if exactly one suitable token is found in the cache.
- FALSE or NA to force the OAuth dance in the browser.

gargle_oob_default

gargle_oob_default() returns TRUE unconditionally on RStudio Server, Posit Workbench, Posit Cloud, or Google Colaboratory, since it is not possible to launch a local web server in these contexts. In this case, for the final step of the OAuth dance, the user is redirected to a specific URL where they must copy a code and paste it back into the R session.

In all other contexts, gargle_oob_default() consults the option named "gargle_oob_default", then the option named "httr_oob_default", and eventually defaults to FALSE.

"oob" stands for out-of-band. Read more about out-of-band authentication in the vignette vignette("auth-from-web").

gargle_oauth_cache

gargle_oauth_cache() returns the option named "gargle_oauth_cache", defaulting to NA. If defined, the option must be set to a logical value or a string:

- TRUE means to cache tokens in the default user-level cache directory, given by rappdirs::user_cache_dir("gargle")
- FALSE means don't cache.
- A string is interpreted as the path to a folder to use as the cache.

The NA default means the default cache directory is used, if it already exists. Otherwise, in an interactive session, gargle asks the user for permission to create the cache. The outcome of this is stored in the "gargle_oauth_cache" option as TRUE or FALSE for the rest of the session.

Use gargle_oauth_sitrep() to see exactly where the cache is and which tokens it holds.

gargle_oauth_client_type

gargle_oauth_client_type() returns the option named "gargle_oauth_client_type", if defined. If defined, the option must be either "installed" or "web". If the option is not defined, the function returns:

- "web" on RStudio Server, Posit Workbench, Posit Cloud, or Google Colaboratory
- "installed" otherwise

Primarily intended to help infer the most suitable OAuth client type when a user is relying on a built-in client, such as the tidyverse client used by packages like bigrquery, googledrive, and googlesheets4.

gargle_verbosity

gargle_verbosity() returns the option named "gargle_verbosity", which determines gargle's verbosity. There are three possible values, inspired by the logging levels of log4j:

- "debug": Fine-grained information helpful when debugging, e.g. figuring out how token_fetch() is working through the registry of credential functions. Previously, this was activated by setting an option named "gargle_quiet" to FALSE.

- "info" (default): High-level information that a typical user needs to see. Since typical gargle usage is always indirect, i.e. gargle is called by another package, gargle itself is very quiet. There are very few messages emitted when `gargle_verbosity = "info"`.
- "silent": No messages at all. However, warnings or errors are still thrown normally.

## Examples

```
gargle_oauth_email()
gargle_oob_default()
gargle_oauth_cache()
gargle_oauth_client_type()
gargle_verbosity()
```

---

`gargle_secret` *Encrypt/decrypt JSON or an R object*

---

## Description

These functions help to encrypt and decrypt confidential information that you might need when deploying gargle-using projects or in CI/CD. They basically rely on inlined copies of the secret functions in the httr2 package. The awkwardness of inlining code from httr2 can be removed if/when gargle starts to depend on httr2.

- The `secret_encrypt_json()` + `secret_decrypt_json()` pair is unique to gargle, given how frequently Google auth relies on JSON files, e.g., service account tokens and OAuth clients.
- The `secret_write_rds()` + `secret_read_rds()` pair is just a copy of functions from httr2. They are handy if you need to secure a user token.
- `secret_make_key()` and `secret_has_key()` are also copies of functions from httr2. Use `secret_make_key` to generate a key. Use `secret_has_key()` to condition on key availability in, e.g., examples, tests, or apps.

## Usage

```
secret_encrypt_json(json, path = NULL, key)

secret_decrypt_json(path, key)

secret_make_key()

secret_write_rds(x, path, key)

secret_read_rds(path, key)

secret_has_key(key)
```

## Arguments

| | |
|---|---|
| json | A JSON file (or string). |
| path | The path to write to (secret_encrypt_json(), secret_write_rds()) or to read from (secret_decrypt_json(), secret_read_rds()). |
| key | Encryption key, as implemented by httr2's secret functions. This should almost always be the name of an environment variable whose value was generated with secret_make_key() (which is an inlined copy of httr2::secret_make_key()). |
| x | An R object. |

## Value

- secret_encrypt_json(): The encrypted JSON string, invisibly. In typical use, this function is mainly called for its side effect, which is to write an encrypted file.
- secret_decrypt_json(): The decrypted JSON string, invisibly.
- secret_write_rds(): x, invisibly
- secret_read_rds(): the decrypted object.
- secret_make_key(): a random string to use as an encryption key.
- secret_has_key() returns TRUE if the key is available and FALSE otherwise.

## Examples

```
# gargle ships with JSON for a fake service account
# here we put the encrypted JSON into a new file
tmp <- tempfile()
secret_encrypt_json(
  fs::path_package("gargle", "extdata", "fake_service_account.json"),
  tmp,
  key = "GARGLE_KEY"
)

# complete the round trip by providing the decrypted JSON to a credential
# function
credentials_service_account(
 scopes = "https://www.googleapis.com/auth/userinfo.email",
 path = secret_decrypt_json(
   fs::path_package("gargle", "secret", "gargle-testing.json"),
   key = "GARGLE_KEY"
 )
)

file.remove(tmp)

# make an artificial Gargle2.0 token
fauxen <- gargle2.0_token(
  email = "jane@example.org",
  client = gargle_oauth_client(
    id = "CLIENT_ID", secret = "SECRET", name = "CLIENT"
  ),
```

```
  credentials = list(token = "fauxen"),
  cache = FALSE
)
fauxen

# store the fake token in an encrypted file
tmp2 <- tempfile()
secret_write_rds(fauxen, path = tmp2, key = "GARGLE_KEY")

# complete the round trip by providing the decrypted token to the "BYO token"
# credential function
rt_fauxen <- credentials_byo_oauth2(
  token  = secret_read_rds(tmp2, key = "GARGLE_KEY")
)
rt_fauxen

file.remove(tmp2)
```

---

`gce_instance_service_accounts` *List all service accounts available on this GCE instance*

---

## Description

List all service accounts available on this GCE instance

## Usage

```
gce_instance_service_accounts()
```

## Value

A data frame, where each row is a service account. Due to aliasing, there is no guarantee that each row represents a distinct service account.

## See Also

The return value is built from a recursive query of the so-called "directory" of the instance's service accounts as documented in https://cloud.google.com/compute/docs/metadata/default-metadata-values#vm_instance_metadata.

## Examples

```
credentials_gce()
```

`init_AuthState` *Create an AuthState*

## Description

Constructor function for objects of class AuthState.

## Usage

```
init_AuthState(
  package = NA_character_,
  client = NULL,
  api_key = NULL,
  auth_active = TRUE,
  cred = NULL,
  app = deprecated()
)
```

## Arguments

| | |
|---|---|
| `package` | Package name, an optional string. It is recommended to record the name of the package whose auth state is being managed. Ultimately, this may be used in some downstream messaging. |
| `client` | A Google OAuth client, preferably constructed via `gargle_oauth_client_from_json()`, which returns an instance of `gargle_oauth_client`. For backwards compatibility, for a limited time, gargle will still accept an "OAuth app" created with `httr::oauth_app()`. |
| `api_key` | Optional. API key (a string). Some APIs accept unauthorized, "token-free" requests for public resources, but only if the request includes an API key. |
| `auth_active` | Logical. `TRUE` means requests should include a token (and probably not an API key). `FALSE` means requests should include an API key (and probably not a token). |
| `cred` | Credentials. Typically populated indirectly via `token_fetch()`. |
| `app` | **[Deprecated]** Replaced by the `client` argument. |

## Value

An object of class AuthState.

## Examples

```
my_client <- gargle_oauth_client(
  id = "some_long_client_id",
  secret = "ssshhhhh_its_a_secret",
  name = "my-nifty-oauth-client"
)
```

```
init_AuthState(
  package = "my_package",
  client = my_client,
  api_key = "api_key_api_key_api_key",
)
```

---

`request_develop` *Build a Google API request*

---

## Description

Intended primarily for internal use in client packages that provide high-level wrappers for users. The `vignette("request-helper-functions")` describes how one might use these functions inside a wrapper package.

## Usage

```
request_develop(
  endpoint,
  params = list(),
  base_url = "https://www.googleapis.com"
)

request_build(
  method = "GET",
  path = "",
  params = list(),
  body = list(),
  token = NULL,
  key = NULL,
  base_url = "https://www.googleapis.com"
)
```

## Arguments

| | |
|---|---|
| `endpoint` | List of information about the target endpoint or, in Google's vocabulary, the target "method". Presumably prepared from the Discovery Document for the target API. |
| `params` | Named list. Values destined for URL substitution, the query, or, for `request_develop()` only, the body. For `request_build()`, body parameters must be passed via the body argument. |
| `base_url` | Character. |
| `method` | Character. An HTTP verb, such as `GET` or `POST`. |

| | |
|---|---|
| path | Character. Path to the resource, not including the API's base_url. Examples: drive/v3/about or drive/v3/files/{fileId}. The path can be a template, i.e. it can include variables inside curly brackets, such as {fileId} in the example. Such variables are substituted by request_build(), using named parameters found in params. |
| body | List. Values to send in the API request body. |
| token | Token, ready for inclusion in a request, i.e. prepared with httr::config(). |
| key | API key. Needed for requests that don't contain a token. For more, see Google's document Credentials, access, security, and identity (https://support.google.com/googleapi/answe A key can be passed as a named component of params, but note that the formal argument key will clobber it, if non-NULL. Either way, the key is sent in the X-goog-api-key request header in order to keep it out of the URL. |

## Value

request_develop(): list() with components method, path, params, body, and base_url.

request_build(): list() with components method, url (the full URL, post-substitution, including the query), body, token, and headers (a named character vector holding the X-goog-api-key header, or NULL).

### request_develop()

Combines user input (params) with information about an API endpoint. endpoint should contain these components:

- path: See documentation for argument.
- method: See documentation for argument.
- parameters: Compared with params supplied by user. An error is thrown if user-supplied params aren't named in endpoint$parameters or if user fails to supply all required parameters. In the return value, body parameters are separated from those destined for path substitution or the query.

The return value is typically used as input to request_build().

### request_build()

Builds a request, in a purely mechanical sense. This function does nothing specific to any particular Google API or endpoint.

- Use with the output of request_develop() or with hand-crafted input.
- params are used for variable substitution in path. Leftover params that are not bound by the path template automatically become HTTP query parameters.
- Adds an API key, in the X-goog-api-key header, if and only if token = NULL and removes the API key otherwise. Client packages should generally pass their own API key in, but note that gargle_api_key() is available for small-scale experimentation.
- Client packages should send the request with request_make() or request_retry(), which apply the headers. Code that makes its own HTTP call with only the url won't send the API key.

See `googledrive::generate_request()` for an example of usage in a client package. googledrive has an internal list of selected endpoints, derived from the Drive API Discovery Document (`https://www.googleapis.com/`) exposed via `googledrive::drive_endpoints()`. An element from such a list is the expected input for `endpoint`. `googledrive::generate_request()` is a wrapper around `request_develop()` and `request_build()` that inserts a googledrive-managed API key and some logic about Team Drives. All user-facing functions use `googledrive::generate_request()` under the hood.

## See Also

Other requests and responses: `request_make()`, `response_process()`

## Examples

```
## Not run:
## Example with a prepared endpoint
ept <- googledrive::drive_endpoints("drive.files.update")[[1]]
req <- request_develop(
  ept,
  params = list(
    fileId = "abc",
    addParents = "123",
    description = "Exciting File"
  )
)
req

req <- request_build(
  method = req$method,
  path = req$path,
  params = req$params,
  body = req$body,
  token = "PRETEND_I_AM_A_TOKEN"
)
req

## Example with no previous knowledge of the endpoint
## List a file's comments
## https://developers.google.com/drive/v3/reference/comments/list
req <- request_build(
  method = "GET",
  path = "drive/v3/files/{fileId}/comments",
  params = list(
    fileId = "your-file-id-goes-here",
    fields = "*"
  ),
  token = "PRETEND_I_AM_A_TOKEN"
)
req

## End(Not run)
```

```
# Example with no previous knowledge of the endpoint and no token
# find books by Hadley Wickham with the Books API,
# using gargle's demo API key (for which the Books API is enabled)
req <- request_build(
  method = "GET",
  path = "books/v1/volumes",
  params = list(
    q = "inauthor:Hadley Wickham",
    maxResults = 10
  ),
  key = gargle_api_key()
)
resp <- request_make(req)
out <- response_process(resp)
books <- lapply(out$items, \(x) x$volumeInfo)
data.frame(
  title = vapply(books, \(x) x$title, character(1)),
  authors = vapply(books, \(x) toString(x$authors), character(1)),
  date = vapply(books, \(x) toString(x$publishedDate), character(1))
)
```

---

`request_make` *Make a Google API request*

---

## Description

Intended primarily for internal use in client packages that provide high-level wrappers for users. `request_make()` does relatively little:

- Calls an HTTP method.
- Adds a user agent.
- Adds any `headers`, such as the API key header added by `request_build()`.
- Enforces `"json"` as the default for `encode`. This differs from httr's default behaviour, but aligns better with Google APIs.

Typically the input is created with `request_build()` and the output is processed with `response_process()`.

## Usage

```
request_make(x, ..., encode = "json", user_agent = gargle_user_agent())
```

## Arguments

| | |
|---|---|
| `x` | List. Holds the components for an HTTP request, presumably created with `request_develop()` or `request_build()`. Must contain a `method` and `url`. If present, `body`, `token`, and `headers` are used. |

| | |
|---|---|
| ... | Optional arguments passed through to the HTTP method. Currently neither gargle nor httr checks that all are used, so be aware that unused arguments may be silently ignored. |
| encode | If the body is a named list, how should it be encoded? Can be one of form (application/x-www-form-urlencoded), multipart, (multipart/form-data), or json (application/json).<br>For "multipart", list elements can be strings or objects created by `upload_file()`. For "form", elements are coerced to strings and escaped, use `I()` to prevent double-escaping. For "json", parameters are automatically "unboxed" (i.e. length 1 vectors are converted to scalars). To preserve a length 1 vector as a vector, wrap in `I()`. For "raw", either a character or raw vector. You'll need to make sure to set the `content_type()` yourself. |
| user_agent | A user agent string, prepared by `httr::user_agent()`. When in doubt, a client package should have an internal function that extends `gargle_user_agent()` by prepending its return value with the client package's name and version. |

## Value

Object of class `response` from httr.

## See Also

Other requests and responses: `request_develop()`, `response_process()`

## Examples

```
## Not run:
req <- gargle::request_build(
  method = "GET",
  path = "path/to/the/resource",
  token = "PRETEND_I_AM_TOKEN"
)
gargle::request_make(req)

## End(Not run)
```

---

# request_retry *Make a Google API request, repeatedly*

---

## Description

Intended primarily for internal use in client packages that provide high-level wrappers for users. It is a drop-in substitute for `request_make()` that also has the ability to retry the request. Codes that are considered retryable: 408, 429, 500, 502, 503.

## Usage

```
request_retry(..., max_tries_total = 5, max_total_wait_time_in_seconds = 100)
```

## Arguments

| | |
|---|---|
| `...` | Passed along to `request_make()`. |
| `max_tries_total` | Maximum number of tries. |
| `max_total_wait_time_in_seconds` | Total seconds we are willing to dedicate to waiting, summed across all tries. This is a technical upper bound and actual cumulative waiting will be less. |

## Details

Consider an example where we are willing to make a request up to 5 times.

```
try  1  2    3        4                5
     |--|----|--------|----------------|
wait  1   2      3           4
```

There will be up to 5 - 1 = 4 waits and we generally want the waiting period to get longer, in an exponential way. Such schemes are called exponential backoff. `request_retry()` implements exponential backoff with "full jitter", where each waiting time is generated from a uniform distribution, where the interval of support grows exponentially. A common alternative is "equal jitter", which adds some noise to fixed, exponentially increasing waiting times.

Either way our waiting times are based on a geometric series, which, by convention, is usually written in terms of powers of 2:

```
b, 2b, 4b, 8b, ...
  = b * 2^0, b * 2^1, b * 2^2, b * 2^3, ...
```

The terms in this series require knowledge of `b`, the so-called exponential base, and many retry functions and libraries require the user to specify this. But most users find it easier to declare the total amount of waiting time they can tolerate for one request. Therefore `request_retry()` asks for that instead and solves for `b` internally. This is inspired by the Opnieuw Python library for retries. Opnieuw's interface is designed to eliminate uncertainty around:

- Units: Is this thing given in seconds? minutes? milliseconds?
- Ambiguity around how things are counted: Are we starting at 0 or 1? Are we counting tries or just the retries?
- Non-intuitive required inputs, e.g., the exponential base.

Let $n$ be the total number of tries we're willing to make (the argument `max_tries_total`) and let $W$ be the total amount of seconds we're willing to dedicate to making and retrying this request (the argument `max_total_wait_time_in_seconds`). Here's how we determine $b$:

```
sum_{i=0}^(n - 1) b * 2^i = W
b * sum_{i=0}^(n - 1) 2^i = W
       b * ( (2 ^ n) - 1) = W
                        b = W / ( (2 ^ n) - 1)
```

## Value

Object of class `response` from httr.

## Special cases

`request_retry()` departs from exponential backoff in three special cases:

- It actually implements *truncated* exponential backoff. There is a floor and a ceiling on random wait times.
- `Retry-After` header: If the response has a header named `Retry-After` (case-insensitive), it is assumed to provide a non-negative integer indicating the number of seconds to wait. If present, we wait this many seconds and do not generate a random waiting time. (In theory, this header can alternatively provide a datetime after which to retry, but we have no first-hand experience with this variant for a Google API.)
- Sheets API quota exhaustion: In the course of googlesheets4 development, we've grown very familiar with the `429 RESOURCE_EXHAUSTED` error. As of 2023-04-15, the Sheets API v4 has a limit of 300 requests per minute per project and 60 requests per minute per user per project. Limits for reads and writes are tracked separately. In our experience, the "60 (read or write) requests per minute per user" limit is the one you hit most often. If we detect this specific failure, the first wait time is a bit more than one minute, then we revert to exponential backoff.

## See Also

- https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- https://www.channable.com/tech/opnieuw-a-simple-and-intuitive-retrying-library-for-python
- https://github.com/channable/opnieuw
- https://docs.cloud.google.com/storage/docs/retry-strategy
- https://www.rfc-editor.org/info/rfc7231/#section-7.1.3
- https://developers.google.com/sheets/api/limits
- https://googleapis.dev/python/google-api-core/latest/retry.html

## Examples

```
## Not run:
req <- gargle::request_build(
  method = "GET",
  path = "path/to/the/resource",
  token = "PRETEND_I_AM_TOKEN"
)
gargle::request_retry(req)

## End(Not run)
```

`response_process` *Process a Google API response*

## Description

`response_process()` is intended primarily for internal use in client packages that provide high-level wrappers for users. Typically applied as the final step in this sequence of calls:

- Request prepared with `request_build()`.
- Request made with `request_make()`.
- Response processed with `response_process()`.

All that's needed for a successful request is to parse the JSON extracted via `httr::content()`. Therefore, the main point of `response_process()` is to handle less happy outcomes:

- Status codes in the 400s (client error) and 500s (server error). The structure of the error payload varies across Google APIs and we try to create a useful message for all variants we know about.
- Non-JSON content type, such as HTML.
- Status code in the 100s (information) or 300s (redirection). These are unexpected.

If `response_process()` results in an error, a redacted version of the `resp` input is returned in the condition (auth tokens are removed).

## Usage

```
response_process(
  resp,
  error_message = gargle_error_message,
  error_class = NULL,
  remember = TRUE,
  call = caller_env()
)

response_as_json(resp, call = caller_env())

gargle_error_message(resp, call = caller_env())
```

## Arguments

| | |
|---|---|
| `resp` | Object of class `response` from httr. |
| `error_message` | Function that produces an informative error message from the primary input, `resp`. It must return a character vector. |
| `error_class` | Optional character vector of error classes to add to the condition object. These classes are prepended to gargle's default classes (`"gargle_error_request_failed"` and `"http_error_{status_code}"`). |

| | |
|---|---|
| remember | Whether to remember the most recently processed response. |
| call | The execution environment of a currently running function, e.g. call = caller_env(). The corresponding function call is retrieved and mentioned in error messages as the source of the error.<br>You only need to supply call when throwing a condition from a helper function which wouldn't be relevant to mention in the message.<br>Can also be NULL or a defused function call to respectively not display any call or hard-code a code to display.<br>For more information about error calls, see Including function calls in error messages. |

## Details

When remember = TRUE (the default), gargle stores the most recently seen response internally, for *post hoc* examination. The stored response is literally just the most recent resp input, but with auth tokens redacted. It can be accessed via gargle_last_response(). A companion function gargle_last_content() returns just the parsed content, which is probably the most useful form for *post mortem* analysis.

The response_as_json() helper is exported only as an aid to maintainers who wish to use their own error_message function, instead of gargle's built-in gargle_error_message(). When implementing a custom error_message function, call response_as_json() immediately on the input in order to inherit gargle's handling of non-JSON input.

## Value

The content of the request, as a list. An HTTP status code of 204 (No content) is a special case returning TRUE.

## See Also

Other requests and responses: request_develop(), request_make()

## Examples

```
## Not run:
# get an OAuth2 token with 'userinfo.email' scope
token <- token_fetch(scopes = "https://www.googleapis.com/auth/userinfo.email")

# see the email associated with this token
req <- gargle::request_build(
  method = "GET",
  path = "v1/userinfo",
  token = token,
  base_url = "https://openidconnect.googleapis.com"
)
resp <- gargle::request_make(req)
response_process(resp)

# make a bad request (this token has incorrect scope)
req <- gargle::request_build(
```

```
  method = "GET",
  path = "fitness/v1/users/{userId}/dataSources",
  token = token,
  params = list(userId = 12345)
)
resp <- gargle::request_make(req)
response_process(resp)

## End(Not run)
```

---

`token-info` *Get info from a token*

---

## Description

These functions send the `token` to Google endpoints that return info about a token or a user.

## Usage

```
token_userinfo(token)

token_email(token)

token_tokeninfo(token)
```

## Arguments

| | |
|---|---|
| `token` | A token with class Token2.0 or an object of httr's class `request`, i.e. a token that has been prepared with `httr::config()` and has a Token2.0 in the `auth_token` component. |

## Details

It's hard to say exactly what info will be returned by the "userinfo" endpoint targetted by `token_userinfo()`. It depends on the token's scopes. Where possible, OAuth2 tokens obtained via the gargle package include the `https://www.googleapis.com/auth/userinfo.email` scope, which guarantees we can learn the email associated with the token. If the token has the `https://www.googleapis.com/auth/userinfo.profile` scope, there will be even more information available. But for a token with unknown or arbitrary scopes, we can't make any promises about what information will be returned.

## Value

A list containing:

- `token_userinfo()`: user info
- `token_email()`: user's email (obtained from a call to `token_userinfo()`)
- `token_tokeninfo()`: token info

## Examples

```
## Not run:
# with service account token
t <- token_fetch(
  scopes = "https://www.googleapis.com/auth/drive",
  path   = "path/to/service/account/token/blah-blah-blah.json"
)
# or with an OAuth token
t <- token_fetch(
  scopes = "https://www.googleapis.com/auth/drive",
  email  = "janedoe@example.com"
)
token_userinfo(t)
token_email(t)
tokens_tokeninfo(t)

## End(Not run)
```

---

`token_fetch` *Fetch a token for the given scopes*

---

## Description

This is a rather magical function that calls a series of concrete credential-fetching functions, each wrapped in a `tryCatch()`. `token_fetch()` keeps trying until it succeeds or there are no more functions to try. See the `vignette("how-gargle-gets-tokens")` for a full description of `token_fetch()`.

## Usage

```
token_fetch(scopes = NULL, ...)
```

## Arguments

| | |
|---|---|
| `scopes` | A character vector of scopes to request. Pick from those listed at https://developers.google.com/identity/protocols/oauth2/scopes.<br>For certain token flows, the `"https://www.googleapis.com/auth/userinfo.email"` scope is unconditionally included. This grants permission to retrieve the email address associated with a token; gargle uses this to index cached OAuth tokens. This grants no permission to view or send email and is generally considered a low-value scope. |
| `...` | Additional arguments passed to all credential functions. |

## Value

An `httr::Token` (often an instance of something that inherits from `httr::Token`) or `NULL`.

## See Also

`cred_funs_list()` reveals the current registry of credential-fetching functions, in order.

Other credential functions: `credentials_app_default()`, `credentials_byo_oauth2()`, `credentials_external_accour`
`credentials_gce()`, `credentials_service_account()`, `credentials_user_oauth2()`

## Examples

```
## Not run:
token_fetch(scopes = "https://www.googleapis.com/auth/userinfo.email")

## End(Not run)
```

# Index


∗ **credential functions**
  credentials_app_default, 5
  credentials_byo_oauth2, 7
  credentials_external_account, 8
  credentials_gce, 10
  credentials_service_account, 12
  credentials_user_oauth2, 13
  token_fetch, 39
∗ **requests and responses**
  request_develop, 29
  request_make, 32
  response_process, 36

AuthState, *28*
AuthState *(*AuthState-class*)*, 2
AuthState-class, 2

content_type(), *33*
cred_funs, 15
cred_funs_add *(*cred_funs*)*, 15
cred_funs_clear *(*cred_funs*)*, 15
cred_funs_list *(*cred_funs*)*, 15
cred_funs_list(), *40*
cred_funs_list_default *(*cred_funs*)*, 15
cred_funs_set *(*cred_funs*)*, 15
cred_funs_set_default *(*cred_funs*)*, 15
credentials_app_default, 5
credentials_app_default(), *8–11*, *13*, *15*, *40*
credentials_byo_oauth2, 7
credentials_byo_oauth2(), *6*, *10, 11*, *13*, *15*, *40*
credentials_external_account, 8
credentials_external_account(), *5, 6*, *8*, *11*, *13*, *15*, *40*
credentials_gce, 10
credentials_gce(), *6*, *8*, *10*, *13*, *15*, *40*
credentials_service_account, 12
credentials_service_account(), *5, 6*, *8*, *10, 11*, *15*, *40*
credentials_user_oauth2, 13
credentials_user_oauth2(), *6*, *8*, *10, 11*, *13*, *40*

defused function call, *37*

field_mask, 18

Gargle2.0, *15*, *19, 20*
gargle2.0_token, *14*, 19
gargle2.0_token(), *13*
gargle_api_key(), *30*
gargle_error_message *(*response_process*)*, 36
gargle_last_content(), *37*
gargle_last_response(), *37*
gargle_oauth_cache *(*gargle_options*)*, 23
gargle_oauth_cache(), *14*, *20*, *22*
gargle_oauth_client *(*gargle_oauth_client_from_json*)*, 20
gargle_oauth_client_from_json, 20
gargle_oauth_client_from_json(), *14*, *19*, *28*
gargle_oauth_client_type *(*gargle_options*)*, 23
gargle_oauth_client_type(), *14*, *20*
gargle_oauth_email *(*gargle_options*)*, 23
gargle_oauth_email(), *14*, *19*
gargle_oauth_sitrep, 22
gargle_oauth_sitrep(), *24*
gargle_oob_default *(*gargle_options*)*, 23
gargle_oob_default(), *14*, *20*
gargle_options, 23
gargle_secret, 25
gargle_verbosity *(*gargle_options*)*, 23
gce_instance_service_accounts, 27
GceToken(), *11*

httr, *33*, *35, 36*

httr::config(), *7*, *30*, *38*
httr::oauth_app(), *14*, *19*, *21*, *28*
httr::Token, *39*
httr::Token2.0, *3*, *6*
httr::TokenServiceAccount, *3*, *6*, *12*
httr::user_agent(), *33*

Including function calls in error messages, *37*
init_AuthState, 28
init_AuthState(), *3, 4*

jsonlite::fromJSON(), *9*, *12*, *21*

local_cred_funs *(*cred_funs*)*, 15
local_gargle_verbosity *(*gargle_options*)*, 23

oauth_app(), *20*
oauth_app_from_json(), *21*

request_build *(*request_develop*)*, 29
request_build(), *32*, *36*
request_develop, 29
request_develop(), *32, 33*, *37*
request_make, 32
request_make(), *30, 31*, *33, 34*, *36, 37*
request_retry, 33
request_retry(), *30*
response_as_json *(*response_process*)*, 36
response_process, 36
response_process(), *31–33*

secret_decrypt_json *(*gargle_secret*)*, 25
secret_encrypt_json *(*gargle_secret*)*, 25
secret_has_key *(*gargle_secret*)*, 25
secret_make_key *(*gargle_secret*)*, 25
secret_read_rds *(*gargle_secret*)*, 25
secret_write_rds *(*gargle_secret*)*, 25
service account token, *13*

token-info, 38
Token2.0, *7, 8*, *38*
token_email *(*token-info*)*, 38
token_fetch, 39
token_fetch(), *3*, *6*, *8–11*, *13*, *15*, *17*, *28*
token_tokeninfo *(*token-info*)*, 38
token_userinfo *(*token-info*)*, 38

upload_file(), *33*

WifToken, *6*
WifToken(), *9*
with_cred_funs *(*cred_funs*)*, 15
with_gargle_verbosity *(*gargle_options*)*, 23
workload identity federation, *13*