

Ghub User and Developer Manual

for version 2.0.1 (v2.0.1-48-g87701ea+1)

Jonas Bernoulli

Copyright (C) 2017-2018 Jonas Bernoulli <jonas@bernoul.li>

You can redistribute this document and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Table of Contents

1	Introduction	1
2	Getting Started.....	2
2.1	Setting the Username.....	2
2.2	Interactively Creating and Storing a Token.....	3
2.3	Manually Creating and Storing a Token.....	5
2.4	How Ghub uses Auth-Source.....	5
3	Using Ghub in Personal Scripts.....	7
4	Using Ghub in a Package.....	8
5	API.....	9
5.1	Making Requests	9
5.2	Authentication	13
5.3	Configuration Variables.....	13
6	GraphQL Support.....	15
7	Support for Other Forges.....	17
7.1	Forge Functions and Variables.....	17
7.2	Forge Limitations and Notes.....	18

1 Introduction

Ghub provides basic support for using the APIs of various Git forges from Emacs packages. Originally it only supported the Github REST API, but now it also supports the Github GraphQL API as well as the REST APIs of Gitlab, Gitea, Gogs and Bitbucket.

Ghub abstracts access to API resources using only a handful of basic functions such as ‘ghub-get’. These are convenience wrappers around ‘ghub-request’. Additional forge-specific wrappers like ‘glab-put’, ‘gtea-put’, ‘gogs-post’ and ‘buck-delete’ are also available. Ghub does not provide any resource-specific functions, with the exception of ‘FORGE-repository-id’.

When accessing Github, then Ghub handles the creation and storage of access tokens using a setup wizard to make it easier for users to get started. The tokens for other forges have to be created manually.

Ghub is intentionally limited to only provide these two essential features — basic request functions and guided setup — to avoid being too opinionated, which would hinder wide adoption. It is assumed that wide adoption would make life easier for users and maintainers alike, because then all packages that talk to forge APIs could be configured the same way.

Fancier interfaces can be implemented on top of Ghub, and one such wrapper — named simply Ghub+ — has already been implemented. The benefit of basing various opinionated interfaces on top of a single library that provides only the core functionality is that choosing the programming interface no longer dictates how access tokens are handled. Users can then use multiple packages that access the Github API without having to learn the various incompatible ways packages expect the appropriate token to be made available to them.

Ghub uses the built-in `auth-source` library to store access tokens. That library is very flexible and supports multiple backends, which means that it is up to the user how secrets are stored. They can, among other things, choose between storing secrets in plain text for ease of use, or encrypted for better security.

Previously (as in until this library is widely adopted) it was up to package authors to decide if things should be easy or secure. (Note that `auth-source` defaults to "easy" — you have been warned.)

Ghub expects package authors to use a dedicated access token instead of sharing a single token between all packages that rely on it. That means that users cannot configure Ghub once and later start using a new package without any additional setup. But Ghub helps with that.

When the user invokes some command that ultimately results in `ghub-request` being called and the appropriate token is not available yet, then the user is guided through the process of creating and storing a new token, and at the end of that process the request is carried out as if the token had been available to begin with.

2 Getting Started

Each package that uses Ghub uses its own token. Despite that, chances are good that after successfully configuring one package you can just start using another package pretty much instantly.

If the necessary token to access a Github instance is not available when a package makes an API request, then a setup wizard pops up, and after answering a few questions you are good to go. Even the request that caused the wizard to be summoned should succeed and for most users this should be true even when configuring the very first token.

However, in some situations some manual configuration is necessary **before** using the wizard, or the wizard cannot be used at all:

- If you don't want to use the wizard then you don't have to and can create tokens manually as described in Section 2.3 [Manually Creating and Storing a Token], page 5.
- Unfortunately only Github supports the creation of tokens by using the API. If you want to access another forge, then you have to create the token manually as describe in Section 2.3 [Manually Creating and Storing a Token], page 5. Also see Chapter 7 [Support for Other Forges], page 17.
- If you want to access a Github Enterprise instance, then you have to tell Ghub about that before the wizard makes its appearance by setting the Git variable `github.host`. You also have to tell Ghub your username for that instance using the variable `github.HOST.user` even if it is the same as on Github.com.
- If the variable `github.user` (or `github.HOST.user` for an Enterprise instance) is unset when the wizard is first summoned, then you are asked to provide your username. That value is then stored **globally** to avoid having to ask you that question once per repository. If you have multiple accounts on Github.com (or a Github Enterprise instance), then you have to explicitly tell Ghub about that. This can be done by setting the repository-local values of the appropriate variable **before** the wizard is invoked.
- You might forget to do the above, which is why it is important to carefully read the output of the wizard. If it turns out that you forgot to set a variable, then you must abort, set the variable, and repeat the request to trigger the wizard again.
- The setup wizard should work even if you have enabled two-factor authentication. However if your Github Enterprise instance enforces Single Sign-On as an additional security measure, then you are out of luck and have to create the token manually as described in Section 2.3 [Manually Creating and Storing a Token], page 5.

The variables mentioned above — and others — are documented in Section 5.3 [Configuration Variables], page 13, and the setup wizard is documented in Section 2.2 [Interactively Creating and Storing a Token], page 3.

2.1 Setting the Username

If you haven't set the Git variable `github.user` yet when making a request, then you will be asked:

```
Git variable 'github.user' is unset. Set to:
```

You are expected to provide your Github username here. The provided value will be saved globally (using `git config --global github.user USERNAME`).

If you need to identify as another user in a particular repository, then you have to set that variable locally, **before** making a request:

```
cd /path/to/repo
git config github.user USERNAME
```

For Github Enterprise instances you have to specify where the API can be accessed **before** you try to access it and a different variable has to be used to set the username. For example if the API is available at `https://example.com/api/v3`, then you should do this:

```
# Do this once
git config --global github.example.com/api/v3.user EMPLOYEE

# Do this for every corporate repository
cd /path/to/repo
git config github.host example.com/api/v3
```

If you do not set `github.example.com/api/v3.user`, then you will be asked to provide the value when trying to make a request, but you do have to manually set `github.host`, or Ghub assumes that you are trying to access `api.github.com`.

2.2 Interactively Creating and Storing a Token

Ghub uses a different token for every package as well as for every machine from which you access the Github API (and obviously also for every Github instance and user). This allows packages to only request the scopes that they actually need and also gives users the opportunity to refuse access to certain scopes if they expect to not use the features that need them.

Usually you don't have to worry about creating and storing a token yourself and can just make a request. Note however that you don't have to use the setup wizard described below. Alternatively you can perform the setup manually as described in the next section.

If you make a request and the required token is not available yet, then the setup wizard will first ask you something like this:

```
Such a Github API token is not available:

Host:    api.github.com
User:    USERNAME
Package: PACKAGE

Scopes requested in 'PACKAGE-github-token-scopes':
  repo
Store on Github as:
  "Emacs package PACKAGE @ LOCAL-MACHINE"
Store locally according to option 'auth-sources':
  ("~/authinfo" "~/authinfo.gpg" "~/netrc")
```

If in doubt, then abort and first view the section of the Ghub documentation called "Manually Creating and Storing a Token".

```
Create and store such a token? (yes or no)
```

If you don't have any doubts, then answer "yes". Lets address some of the doubts that you might have:

- `Host` usually is "api.github.com" and that is usually what you want. If you are trying to access a Github Enterprise instance, then it should be something else and you have to set the value manually before the setup wizard is summoned, as described in the parent section.
- `User` should be your Github.com (or Github Enterprise instance) username. If it is something else and it doesn't look like a simple typo, then you should read the parent section again. In either case you have to abort.

- `Package` should be the name of the package you are using to access the Github API. If it is `ghub`, then the package author disregarded that convention and you should probably report a bug in the issue tracker of that package.

Or you yourself are using `ghub-request` or one of its wrappers directly, in which case this is expected and perfectly fine. In that case you might however want to abort and change the value of the variable `ghub-github-token-scopes` before triggering the wizard again.

- Each `PACKAGE` has to specify the tokens that it needs using a variable named `PACKAGE-github-token-scopes`. The doc-string of that variable should document why the various scopes are needed.

The meaning of the various scopes are documented at <https://magit.vc/goto/f63aeb0a>.

- The value of `auth-sources` is shown. The default value causes secrets to be stored in plain text. Because this might be unexpected, Ghub additionally displays a warning when appropriate.

```
WARNING: The token will be stored unencrypted in "~/authinfo".
        If you don't want that, you have to abort and customize
        the 'auth-sources' option.
```

Whether that is something that needs fixing, is up to you. If your answer is yes, then you should abort and see Section 2.4 [How Ghub uses Auth-Source], page 5, for instructions on how to save the token more securely.

- When creating a token it is necessary to provide a token description. Ghub uses descriptions that have the form "Emacs package PACKAGE @ LOCAL-MACHINE". Github uses the token description to identify the token, not merely as something useful to humans. Token descriptions therefore have to be unique and in rare cases you get an additional prompt, asking you something like:

```
A token named "Emacs package PACKAGE @ LOCAL-MACHINE"
already exists on Github. Replace it?
```

You might see this message when you have lost the old token and want to replace it with a new one, in which case you should obviously just proceed.

Or two of your computers have the same hostname, which is bad practice because it gains you nothing but leads to issues such as this. Or you are dual-booting on this machine and use the same hostname in all operating systems, which is a somewhat reasonable thing to do, but never-the-less leads to issues like this.

In either case you will have to use something other than the value returned by `system-name` to identify the current machine or operating system. Or you can continue to identify different things using the same identifier, in which case you have to manually distribute the token.

The former is recommended and also easier to do, using the variable `ghub-override-system-name`. See Section 5.3 [Configuration Variables], page 13, for details.

After the above prompt you are also asked for your username and password. If you have enabled two-factor authentication, then you also have to provide the authentication code at least twice. If you make sure the code is still good for a while when asked for it first, then you can just press RET at the later prompt(s).

2.3 Manually Creating and Storing a Token

If you cannot or don't want to use the wizard then you have to (1) figure out what scopes a package wants, (2) create such a token using the web interface and (3) store the token where Ghub expects to find it.

A package named `PACKAGE` has to specify the scopes that it wants in the variable named `PACKAGE-ghub-token-scopes`. The doc-string of such variables should document what the various scopes are needed for.

To create or edit a token go to <https://github.com/settings/tokens>. For Gitlab.com use https://gitlab.com/profile/personal_access_tokens.

Finally store the token in a place where Ghub looks for it, as described in Section 2.4 [How Ghub uses Auth-Source], page 5.

If you store the token in a file like `~/.authinfo`, then note that `auth-source`'s parsing of that file is brittle. Make sure the file ends with a newline character, that there are no empty or invalid lines, and that all comments are prefixed with `#`.

2.4 How Ghub uses Auth-Source

Please see `auth` for all the gory details about Auth-Source. Some Ghub-specific information and important notes follow.

The variable `auth-sources` controls how and where Auth-Source stores new secrets and where it looks for known secrets. The default value is ("`~/.authinfo`" "`~/.authinfo.gpg`" "`~/.netrc`"), which means that it looks in all of these files in order to find secrets and that it stores new secrets in `~/.authinfo` because that is the first element of the list. It doesn't matter which files already do or don't exist when storing a new secret, the first file is always used.

Secrets are stored in `~/.authinfo` in plain text. If you don't want that (good choice), then you have to customize `auth-sources`, e.g. by flipping the positions of the first two elements.

Auth-Source also supports storing secrets in various key-chains. Refer to its documentation for more information.

Some Auth-Source backends only support storing three values per entry, the "machine", the "login" and the "password". Because Ghub uses separate tokens for each

package, it has to squeeze four values into those three slots, and it does that by using "USERNAME^PACKAGE" as the "login".

Assuming your username is "ziggy", the package is named "stardust", and you want to access **Github.com** an entry in one of the three mentioned files would then look like this:

```
machine api.github.com login ziggy^stardust password 012345abcdef...
```

Assuming your username is "ziggy", the package is named "stardust", and you want to access **Gitlab.com** an entry in one of the three mentioned files would then look like this:

```
machine gitlab.com/api/v4 login ziggy^stardust password 012345abcdef...
```

3 Using Ghub in Personal Scripts

You can use `ghub-request` and its wrapper functions in your personal scripts, of course. Unlike when you use Ghub from a package that you distribute for others to use, you don't have to specify a package in personal scripts.

```
;; This is perfectly acceptable in personal scripts ...
(ghub-get "/user")
```

```
;; ... and actually equal to
(ghub-get "/user" nil :auth 'ghub)
```

```
;; In packages you have to specify the package using AUTH.
(ghub-get "/user" nil :auth 'foobar)
```

When you do not specify the `AUTH` argument, then a request is made on behalf of the `ghub` package itself. Like for any package that uses Ghub, `ghub` has to declare what scopes it needs, using, in this case, the variable `ghub-github-token-scopes`.

The default value of that variable is `(repo)` and you might want to add additional scopes. You can later add additional scopes to an existing token, using the web interface at <https://github.com/settings/tokens>.

If you do that, then you might want to also set the variable accordingly, but note that Ghub only consults that when **creating** a new token. If you want to know a token's effective scopes use the command `ghub-token-scopes`, described in the next section.

4 Using Ghub in a Package

Every package should use its own token. This allows you as the author of some package to only request access to API scopes that are actually needed, which in turn might make it easier for users to trust your package not to do unwanted things.

The scopes used by `PACKAGE` have to be defined using the variable `PACKAGE-github-token-scopes`, and you have to tell `ghub-request` on behalf of which package a request is being made by passing the symbol `PACKAGE` as the value of its `AUTH` argument.

```
(ghub-request "GET" "/user" nil :auth 'PACKAGE)
```

`PACKAGE-github-token-scopes` [Variable]

This variable defines the token scopes requested by the package named `PACKAGE`. The doc-string should explain what the various scopes are needed for to prevent users from giving `PACKAGE` fewer permissions than it absolutely needs and also to give them greater confidence that `PACKAGE` is only requesting the permissions that it actually needs.

The value of this variable does not necessarily correspond to the scopes that the respective token actually gives access to. There is nothing that prevents users from changing the value **after** creating the token or from editing the token's scopes later on.

So it is pointless to check the value of this variable before making a request. You also should not query the API to reliably determine the supported tokens before making a query. Doing the latter would mean that every request becomes two requests and that the first request would have to be done using the user's password instead of a token.

`ghub-token-scopes` [Command]

Because we cannot be certain that the user hasn't messed up the scopes, Ghub provides this command to make it easy to debug such issues without having to rely on users being thoughtful enough to correctly determine the used scopes manually.

Just tell users to run `M-x ghub-token-scopes` and to provide the correct values for the `HOST`, `USERNAME` and `PACKAGE` when prompted, and to then post the output.

It is to be expected that users will occasionally mess that up so this command outputs not only the scopes but also the user input so that you can have greater confidence in the validity of the user's answer.

```
Scopes for USERNAME^PACKAGE@HOST: (SCOPE...)
```

5 API

This section describes the Ghub API. In other words it describes the public functions and variables provided by the Ghub package and not the APIs of the supported forges, which can be accessed by using those functions. The forge APIs are documented at:

- Github: <https://developer.github.com/v3>
- Gitlab: <https://docs.gitlab.com/ee/api/README.html>
- Gitea: <https://docs.gitea.io/en-us/api-usage> and <https://try.gitea.io/api/swagger>
- Gogs: <https://github.com/gogs/go-gogs-client/wiki>
- Bitbucket: <https://developer.atlassian.com/bitbucket/api/2/reference>

5.1 Making Requests

`ghub-request` *method resource* **&optional** *params &key query* [Function]
payload headers unpaginate noerror reader username auth host callback
*errorback url value error extra method**

This function makes a request for `RESOURCE` using `METHOD`. `PARAMS`, `QUERY`, `PAYLOAD` and/or `HEADERS` are alists holding additional request data. The response body is returned and the response header is stored in the variable `ghub-response-headers`.

- `METHOD` is the HTTP method, given as a string.
- `RESOURCE` is the resource to access, given as a string beginning with a slash.
- `PARAMS`, `QUERY`, `PAYLOAD` and `HEADERS` are alists and are used to specify request data. All these arguments are alists that resemble the JSON expected and returned by the Github API. The keys are symbols and the values stored in the `cdr` (not the `cadr`) can be strings, integers, or lists of strings and integers.

The Github API documentation is vague on how data has to be transmitted and for a particular resource usually just talks about "parameters". Generally speaking when the `METHOD` is "HEAD" or "GET", then they have to be transmitted as a query, otherwise as a payload.

- Use `PARAMS` to automatically transmit like `QUERY` or `PAYLOAD` would depend on `METHOD`.
- Use `QUERY` to explicitly transmit data as a query.
- Use `PAYLOAD` to explicitly transmit data as a payload. Instead of an alist, `PAYLOAD` may also be a string, in which case it gets encoded as UTF-8 but is otherwise transmitted as-is.
- Use `HEADERS` for those rare resources that require that the data is transmitted as headers instead of as a query or payload. When that is the case, then the Github API documentation usually mentions it explicitly.
- If `SILENT` is non-nil, then progress reports and the like are not messaged.
- If `UNPAGINATE` is `t`, then this function make as many requests as necessary to get all values. If `UNPAGINATE` is a natural number, then it gets at most that many pages. For any other non-nil value it raises an error.

- If `NOERROR` is non-`nil`, then no error is raised if the request fails and `nil` is returned instead. If `NOERROR` is `return`, then the error payload is returned instead of `nil`.
- If `READER` is non-`nil`, then it is used to read and return from the response buffer. The default is `ghub--read-json-payload`. For the very few resources that do not return JSON, you might want to use `ghub--decode-payload`.
- If `USERNAME` is non-`nil`, then the request is made on behalf of that user. It is better to specify the user using the Git variable `github.user` for "api.github.com", or `github.HOST.user` if connecting to a Github Enterprise instance.
- Each package that uses Ghub should use its own token. If `AUTH` is `nil` or unspecified, then the generic `ghub` token is used instead. This is only acceptable for personal utilities. A packages that is distributed to other users should always use this argument to identify itself, using a symbol matching its name.

Package authors who find this inconvenient should write a wrapper around this function and possibly for the method-specific functions as well.

Beside `nil`, some other symbols have a special meaning too. `none` means to make an unauthorized request. `basic` means to make a password based request. If the value is a string, then it is assumed to be a valid token. `basic` and an explicit token string are only intended for internal and debugging uses.

If `AUTH` is a package symbol, then the scopes are specified using the variable `AUTH-github-token-scopes`. It is an error if that is not specified. See `ghub-github-token-scopes` for an example.

- If `HOST` is non-`nil`, then connect to that Github instance. This defaults to "api.github.com". When a repository is connected to a Github Enterprise instance, then it is better to specify that using the Git variable `github.host` instead of using this argument.
- If `FORGE` is `gitlab`, then connect to Gitlab.com or, depending on `HOST`, to another Gitlab instance. This is only intended for internal use. Instead of using this argument you should use function `glab-request` and other `glab-*` functions.
- If `CALLBACK` and/or `ERRORBACK` is non-`nil`, then this function makes one or more asynchronous requests and calls `CALLBACK` or `ERRORBACK` when finished. If an error occurred, then it calls `ERRORBACK`, or if that is `nil`, then `CALLBACK`. When no error occurred then it calls `CALLBACK`. When making asynchronous requests, then no errors are signaled, regardless of the value of `NOERROR`.

Both callbacks are called with four arguments.

- For `CALLBACK`, the combined value of the retrieved pages. For `ERRORBACK`, the error that occurred when retrieving the last page.
- The headers of the last page as an alist.
- Status information provided by `url-retrieve`. Its `:error` property holds the same information as the first argument to `ERRORBACK`.
- A `ghub--req` struct, which can be passed to `ghub-continue` (which see) to retrieve the next page, if any.

`ghub-continue` *args*

[Function]

If there is a next page, then this function retrieves that.

This function is only intended to be called from callbacks. If there is a next page, then that is retrieved and the buffer that the result will be loaded into is returned, or `t` if the process has already completed. If there is no next page, then return `nil`.

Callbacks are called with four arguments (see `ghub-request`). The fourth argument is a `ghub--req` struct, intended to be passed to this function. A callback may use the struct's `extra` slot to pass additional information to the callback that will be called after the next request. Use the function `ghub-req-extra` to get and set the value of that slot.

As an example, using `ghub-continue` in a callback like so:

```
(ghub-get "/users/tarsius/repos" nil
  :callback (lambda (value _headers _status req)
              (unless (ghub-continue req)
                    (setq my-value value))))
```

is equivalent to:

```
(ghub-get "/users/tarsius/repos" nil
  :unpaginate t
  :callback (lambda (value _headers _status _req)
              (setq my-value value)))
```

To demonstrate how to pass information from one callback to the next, here we record when we start fetching each page:

```
(ghub-get "/users/tarsius/repos" nil
  :extra (list (current-time))
  :callback (lambda (value _headers _status req)
              (push (current-time) (ghub-req-extra req))
              (unless (ghub-continue req)
                    (setq my-times (ghub-req-extra req))
                    (setq my-value value))))
```

ghub-response-headers [Variable]

A select few Github API resources respond by transmitting data in the response header instead of in the response body. Because there are so few of these inconsistencies, `ghub-request` always returns the response body.

To access the response headers use this variable after `ghub-request` has returned.

ghub-response-link-relations *req headers payload* [Function]

This function returns an alist of the link relations in `HEADERS`, or if optional `HEADERS` is `nil`, then those in `ghub-response-headers`.

When accessing a Bitbucket instance then the link relations are in `PAYLOAD` instead of `HEADERS`, making their API merely RESTish and forcing this function to append those relations to the value of `ghub-response-headers`, for later use when this function is called with `nil` for `PAYLOAD`.

ghub-override-system-name [Variable]

If non-`nil`, the value of this variable is used to override the value returned by `system-name` for the purpose of identifying the local machine, which is necessary because Github uses separate tokens for each machine. Also see Section 5.3 [Configuration Variables], page 13.

`ghub-github-token-scopes` [Variable]

`PACKAGE-github-token-scopes` [Variable]

Such a variable defines the token scopes requested by the respective package `PACKAGE` given by the first word in the variable name. `ghub` itself is treated like any other package. Also see Chapter 4 [Using Ghub in a Package], page 8.

`ghub-head resource &optional params &key query payload headers` [Function]
unpaginate noerror reader username auth host callback errorback

`ghub-get resource &optional params &key query payload headers` [Function]
unpaginate noerror reader username auth host callback errorback

These functions are simple wrappers around `ghub-request`. Their signature is identical to that of the latter, except that they do not have an argument named `METHOD`. The HTTP method is instead given by the second word in the function name.

As described in the documentation for `ghub-request`, it depends on the used method whether the value of the `PARAMS` argument is used as the query or the payload. For the "HEAD" and "GET" methods it is used as the query.

`ghub-put resource &optional params &key query payload headers` [Function]
unpaginate noerror reader username auth host callback errorback

`ghub-post resource &optional params &key query payload headers` [Function]
unpaginate noerror reader username auth host callback errorback

`ghub-patch resource &optional params &key query payload headers` [Function]
unpaginate noerror reader username auth host callback errorback

`ghub-delete resource &optional params &key query payload headers` [Function]
unpaginate noerror reader username auth host callback errorback

These functions are simple wrappers around `ghub-request`. Their signature is identical to that of the latter, except that they do not have an argument named `METHOD`. The HTTP method is instead given by the second word in the function name.

As described in the documentation for `ghub-request`, it depends on the used method whether the value of the `PARAMS` argument is used as the query or the payload. For the "PUT", "POST", "PATCH" and "DELETE" methods it is used as the payload.

`ghub-wait resource &optional duration &key username auth host` [Function]

Some API requests result in an immediate successful response even when the requested action has not actually been carried out yet. An example is the request for the creation of a new repository, which doesn't cause the repository to immediately become available. The Github API documentation usually mentions this when describing an affected resource.

If you want to do something with some resource right after making a request for its creation, then you might have to wait for it to actually be created. This function can be used to do so. It repeatedly tries to access the resource until it becomes available or until the timeout exceeds. In the latter case it signals `ghub-error`.

`RESOURCE` specifies the resource that this function waits for.

`DURATION` specifies the maximum number of seconds to wait for, defaulting to 64 seconds. Emacs will block during that time, but the user can abort using `C-g`.

The first attempt is made immediately and will often succeed. If not, then another attempt is made after two seconds, and each subsequent attempt is made after waiting as long as we already waited between all preceding attempts combined.

See `ghub-request`'s documentation above for information about the other arguments.

`ghub-graphql` *graphql* **&optional** *variables* **&key** *username* *auth* *host* [Function]
callback

This function makes a GraphQL request using `GRAPHQL` and `VARIABLES` as inputs. `GRAPHQL` is a GraphQL string. `VARIABLES` is a JSON-like alist. The other arguments behave as for `ghub-request` (which see).

The response is returned as a JSON-like alist. Even if the response contains `errors`, this function does not raise an error. Cursor-handling is likewise left to the caller.

5.2 Authentication

`ghub-create-token` [Command]

This command creates a new token using the values it reads from the user and then stores it according to the variable `auth-sources`. It can also be called non-interactively, but you shouldn't do that yourself.

This is useful if you want to fully setup things before attempting to make the initial request, if you want to provide fewer than the requested scopes or customize `auth-sources` first, or if something has gone wrong when using the wizard that is used when making a request without doing this first. (Note that instead of using this command you can also just repeat the initial request after making the desired adjustments — that is easier.)

This command reads, in order, the `HOST` (Github instance), the `USERNAME`, the `PACKAGE`, and the `SCOPES` in the minibuffer, providing reasonable default choices. `SCOPES` defaults to the scopes that `PACKAGE` requests using the variable `PACKAGE-github-token-scopes`.

`ghub-token-scopes` [Command]

Users are free to give a token access to fewer scopes than what the respective package requested. That can, of course, lead to issues, and package maintainers have to be able to quickly determine if such a (mis-)configuration is the root cause when users report issues.

This command reads the required values in the minibuffer and then shows a message containing these values along with the scopes of the respective token. It also returns the scopes (only) when called non-interactively. Also see Chapter 4 [Using Ghub in a Package], page 8.

5.3 Configuration Variables

The username and, unless you only use Github.com itself, the Github Enterprise instance have to be configured using Git variables. In rare cases it might also be necessary to specify the identity of the local machine, which is done using a lisp variable.

`github.user` [Variable]

The Github.com username. This should be set globally and if you have multiple Github.com user accounts, then you should set this locally only for those repositories that you want to access using the secondary identity.

`github.HOST.user` [Variable]

This variable serves the same purpose as `github.user` but for the Github Enterprise instance identified by `HOST`.

The reason why separate variables are used is that this makes it possible to set both values globally instead of having to set one of the values locally in each and every repository that is connected to the Github Enterprise instance, not Github.com.

`github.host` [Variable]

This variable should only be set locally for a repository and specifies the Github Enterprise edition that that repository is connected to. You should not set this globally because then each and every repository becomes connected to the specified Github Enterprise instance, including those that should actually be connected to Github.com.

When this is undefined, then "api.github.com" is used (defined in the constant `ghub-default-host`, which you should never attempt to change.)

`ghub-override-system-name` [Variable]

Ghub uses a different token for each quadruple (`USERNAME PACKAGE HOST LOCAL-MACHINE`). Theoretically it could reuse tokens to some extent but that would be more difficult to implement, less flexible, and less secure (though slightly more convenient).

A token is identified on the respective Github instance (Github.com or a Github Enterprise instance) using the pair (`PACKAGE . LOCAL-MACHINE`), or more precisely the string "Emacs package `PACKAGE @ LOCAL-MACHINE`". `USERNAME` and `HOST` do not have to be encoded because the token is stored for `USERNAME` on `HOST` and cannot be used by another user and/or on another instance.

There is one potential problem though; for any given (`PACKAGE . LOCAL-MACHINE`) there can only be one token identified by "Emacs package `PACKAGE @ LOCAL-MACHINE`"; Github does not allow multiple tokens with the same description because it uses the description as the identifier (it could use some hash instead, but alas it does not).

If you have multiple machines and some of them have the same name, then you should probably change that as this is not how things ought to be. However if you dual-boot, then it might make sense to give that machine the same name regardless of what operating system you have booted into.

You could use the same token on both operating systems, but setting that up might be somewhat difficult because it is not possible to download an existing token from Github. You could, of course, locally copy the token, but that is inconvenient and would make it harder to only revoke the token used on your infected Windows installation without also revoking it for your totally safe *BSD installation.

Alternatively you can set this variable to a unique value, that will then be used to identify the local machine instead of the value returned by `system-name`.

6 GraphQL Support

`ghub-graphql` *graphql &optional variables &key username auth host* [Function]
callback silent callback errorback value extra

This function makes a GraphQL request using `GRAPHQL` and `VARIABLES` as inputs. `GRAPHQL` is a GraphQL string. `VARIABLES` is a JSON-like alist. The other arguments behave as for `ghub-request` (which see).

The response is returned as a JSON-like alist. Even if the response contains `errors`, this function does not raise an error. Cursor-handling is likewise left to the caller.

`ghub-graphql` is a thin convenience wrapper around `ghub-request`, similar to `ghub-post` and `friends`. While the latter only hard-code the value of the `METHOD` argument, the former also hard-codes `RESOURCE` and constructs `PAYLOAD` from `GRAPHEQL` and `VARIABLES`. It also drops `UNPAGINATE`, `NOERROR`, `READER` (internal functions expect alist-ified JSON) and `FORGE` (only Github currently supports GraphQL).

`ghub-graphql` does not account for the fact that pagination works differently in GraphQL than it does in REST, so users of this function have to deal with that themselves. Likewise error handling works differently and has to be done by the caller too.

An early attempt at implementing automatic unpaging for GraphQL can be found in the `faithful-graphql` branch, provided I haven't deleted that by now. On that branch I try to do things as intended by the designers of GraphQL, using variables and fragments, and drowning in a sea of boilerplate.

The problem with that approach is that it only works for applications that fetch specific information on demand and actually want things to be paginated. I am convinced that GraphQL is very nice for web apps.

However the Forge package for which I am implementing all of this has very different needs. It wants to fetch "all the data" and "cache" it locally, so that it is available even when there is no internet connection. GraphQL was designed around the idea that you should be able to "ask for what you need and get exactly that". But when that boils down to "look, if I persist, then you are going to hand me over all the data anyway, so just caught it up already", then things start to fall apart. If Github's GraphQL allowed pagination to be turned off completely, then teaching `ghub-graphql` about error handling would be enough.

But it doesn't and when doing things as intended, then that leads to huge amounts of repetitive boilerplate, which is so boring to write that doing it without introducing bugs left and right is near impossible; so I decided to give up on GraphQL variables, fragments and conditions, and instead implement something more powerful, though also more opinionated.

`ghub--graphql-vacuum` *query variables callback &optional until* [Function]
&key narrow username auth host forge

This function is an opinionated alternative to `ghub-graphql`. It relies and dark magic to get the job done.

It makes an initial request using `QUERY`. It then looks for paginated edges in the returned data and makes more requests to resolve them. In order to do so it automatically transforms the initial `QUERY` into another query suitable for that particular

edge. The data retrieved by subsequent requests is then injected into the data of the original request before that is returned or passed to the callback. If subsequently retrieved data features new paginated edges, then those are followed recursively.

The end result is essentially the same as using `ghub-graphql`, if only it were possible to say "do not paginate anything". The implementation is much more complicated because it is not possible to do that.

`QUERY` is a GraphQL query expressed as an s-expression. The `graphql` package is used to turn that into a GraphQL query string, but the format is somewhat different than as documented for that package. Also only a subset of the GraphQL features are supported; fragments for example are not, and magical stuff happens to variables. This is not documented yet, I am afraid. Look at existing callers.

`VARIABLES` is a JSON-like alist as for `ghub-graphql`.

`UNTIL` is an alist (`((EDGE-until . VALUE)...`). When unpaginating `EDGE` try not to fetch beyond the element whose first field has the value `VALUE` and remove that element as well as all "lesser" elements from the retrieved data if necessary. Look at `forge--pull-repository` for an example. This is only useful if you "cache" the response locally and want to avoid fetching data again that you already have.

Other arguments behave as for `ghub-graphql` and `ghub-request`, more or less.

Using `ghub--graphql-vacuum`, the following resource specific functions are implemented. These functions are not part of the public API yet and are very much subject to change.

`ghub-fetch-repository` *owner name callback &optional until &key* [Function]
username auth host forge

This function asynchronously fetches forge data about the specified repository. Once all data has been collected, `CALLBACK` is called with the data as the only argument.

`ghub-fetch-issue` *owner name callback &optional until &key* [Function]
username auth host forge

This function asynchronously fetches forge data about the specified issue. Once all data has been collected, `CALLBACK` is called with the data as the only argument.

`ghub-fetch-pullreq` *owner name callback &optional until &key* [Function]
username auth host forge

This function asynchronously fetches forge data about the specified pull-request. Once all data has been collected, `CALLBACK` is called with the data as the only argument.

Note that in order to avoid duplication all of these functions base their initial query on the query stored in `ghub-fetch-repository`. The latter two pass that query through `ghub--graphql-prepare-query`, which then used `ghub--graphql-narrow-query` to remove parts the caller is not interested in. These two functions are also used internally, when unpaginating, but as demonstrated here they can be useful even before making an initial request.

7 Support for Other Forges

7.1 Forge Functions and Variables

Originally Ghub supported only Github but now it also supports Gitlab, Gitea, Gogs and Bitbucket. The function `ghub-request` and all the `ghub-METHOD` convenience wrappers default to acting on a Github forge but can be told to act on another forge using their `FORGE` argument.

The `FORGE` argument only specifies what kind of forge to act on, not which instance. The `HOST` argument can be used to select the instance. For some forges a default instance is defined:

- Forge `github` defaults to host `api.github.com`.
- Forge `gitlab` defaults to host `gitlab.com/api/v4`.
- Forge `bitbucket` defaults to host `api.bitbucket.org/2.0`.
- No canonical host exists for the `gitea` and `gogs` forges and `localhost:3000/api/v1` is used as the default host in both cases.

Together the `FORGE` and `HOST` arguments specify the forge type and instance. In addition to that, it is also necessary to specify on whose behalf the request is being made, which can be done using the `USERNAME` and `AUTH` arguments.

Having to specify these arguments for every request is inconvenient. Additional variables and convenience functions can be used to make that unnecessary in most cases.

These variables can be set globally and/or for a specific repository as explained in Section 5.3 [Configuration Variables], page 13, with a focus on Github instances. To summarize:

- For `https://github.com` the Git variable `github.user` specifies the user.
- For another `github` instance the Git variable `github.HOST.user` specifies the user. The `HOST` in that variable name is the same as the value of the `HOST` argument of the called function.
- Instead of specifying the `HOST` in every function call, the Git variable `github.host` can be used. This should only be set locally.

For `gitlab` and `bitbucket` forges similar variables are available:

- `gitlab.user` specifies the `https://gitlab.com` user.
- `gitlab.HOST.user` specifies the user for the `HOST gitlab` instance.
- `gitlab.host` specifies the `gitlab` host, unless the `HOST` argument is non-nil
- `bitbucket.user` specifies the `https://bitbucket.org` user.
- `bitbucket.HOST.user` specifies the user for the `HOST bitbucket` instance.
- `bitbucket.host` specifies the `bitbucket` host, unless the `HOST` argument is non-nil.

For `gitea` and `gogs` forges some similar variables are available, however for some of the `ghub.*` variables no equivalent variable exist for these two forges:

- `gitea.user` is **not** used because no canonical `gitea` instance exists.
- `gitea.HOST.user` specifies the user for the `HOST gitea` instance.

- `gitea.host` specifies the `gitea` host, unless the `HOST` argument is non-nil
- `gogs.user` is **not** used because no canonical `gitea` instance exists.
- `gogs.HOST.user` specifies the user for the `HOST` `gogs` instance.
- `gogs.host` specifies the `gogs` host, unless the `HOST` argument is non-nil

`ghub-request` and `ghub-METHOD` can be used to make a request for any of the supported forge types, but except when making a request for a `github` instance, then that requires the use of the `FORGE` argument.

To avoid that, functions named `FORGE-request` and `FORGE-METHOD` are also available. The following forms are equivalent, for example:

```
(ghub-get ... :auth 'PACKAGE :forge 'gitlab)
(glab-get ... :auth 'PACKAGE)
```

These forms would remain equivalent even if you did not specify a value for the `AUTH` arguments — but you should not do that if you plan to share your code with others (see Chapter 4 [Using Ghub in a Package], page 8). If you do omit `AUTH`, then the request is made on behalf of the `ghub` package, **regardless** of the symbol prefix of the function you use to do so.

All `FORGE-request` and `FORGE-METHOD` functions, including but not limited to `ghub-METHOD`, are very simple wrappers around `ghub-request`. They take fewer arguments than `ghub-request` and instead pass constant values for the arguments `METHOD` and/or `FORGE`.

7.2 Forge Limitations and Notes

- The token creation wizard is only available for `github` forges, because all other forges do not support using the API to create an API token. As a consequence, if the user makes a request and the necessary token cannot be found, then that results in an error. Tokens can be created at:
 - Gitlab: https://gitlab.com/profile/personal_access_tokens
 - Bitbucket: <https://bitbucket.org/account/user/tarsius/app-passwords>
 - Gitea: <https://localhost:3000/user/settings/applications>
 - Gogs: <https://localhost:3000/user/settings/applications>

Also see Section 2.3 [Manually Creating and Storing a Token], page 5, and Section 2.4 [How Ghub uses Auth-Source], page 5.

- As mentioned in the previous node, the variables `gitea.host` and `gogs.host` are not taken into account.
- Gitea and Gogs do not support limiting a token to certain scopes.
- The Bitbucket API is fairly broken. Some resources only work if a slash is appended while others only work if no slash is appended. I am unable to access any private repositories and some resources don't work for me at all. Also the API is only RESTish; pagination information is part of the response body instead of the header. Due to such issues it is possible that I will eventually have to remove support for Bitbucket altogether.

- The Gitlab API documentation is not always accurate, though I don't have an example at hand. It also isn't structured well, making it occasionally difficult to find the information one is looking for.
- Where one would use `user/repo` when accessing another forge, one has to use `user%2Frepo` when accessing Gitlab, e.g.:

```
(glab-get "/projects/python-mode-devs%2Fpython-mode")
```