Ghub User and Developer Manual

for version 5.0.4

Jonas Bernoulli

Table of Contents

1 Introduction

Ghub is an Emacs library that is used by various packages to access the APIs of various instances of various Git forge implementations.

A forge is a web-based collaborative software platform for developing and distributing computer applications. Examples include Github and Gitlab.

2 Getting Started

This manual guides you through the steps that are necessary to use the Forge package and/or to make a request using just Ghub itself, such as this:

```
(ghub-request "GET" "/user")
```

To be able to do that, Ghub needs to know who you want to talk to, who you are, and how you are going to prove the latter to the former.

Additionally Ghub wants to know on behalf of which Emacs package it is making a request. So the question of "who is making the request" has to be rephrased as "which human (or bot) is using what (Emacs) package to make the request". If, for example, the human known as "tarsius" is using the `forge` package, then that is represented in some places using the string "tarsius$^{f}orge$".

2.1 Basic Concepts, Arguments and Variables

(If you are trying to configure Forge for `https://github.com` and/or `https://gitlab.com`, then you can skip this section.)

Originally Ghub supported only Github but now it also supports Gitlab, Gitea, Gogs and Bitbucket. Because of that, the function `ghub-request` defaults to acting on a `github` forge, but can be told to act on other forges using the FORGE argument.

The FORGE argument only specifies what kind of forge we are accessing, not which host specifically. 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. For example:

```
(ghub-request "GET" "/user" nil
              :forge 'github
              :host "api.github.com"
              :username "tarsius"
              :auth 'forge)
```

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.

(But for debugging purposes the above explicit form is very useful. You will obviously have to change the value of USERNAME and you should use `ghub` as AUTH when first trying this at home.)

These variables can be set globally and/or for a specific repository.

- For "api.github.com" (aka the API of `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.

These `github` specific variables are discussed in more detail in Section 2.4 [Github Configuration Variables], page 5.

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 the `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 `gogs` 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`

2.2 Setting the Username

Ghub needs to know your username, which you use on the host that you want it to connect to. For each host a different Git variable has to be set, to specify the username on that host. More than one variable is needed, because you might use different usernames on different hosts.

Setting your Github.com Username

To inform Ghub about your "github.com" username do this:

```
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:

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

Setting your Gitlab.com Username

To inform Ghub about your "gitlab.com" username do this:

```
git config --global gitlab.user USERNAME
```

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

```
cd /path/to/repo
git config --local gitlab.user USERNAME
```

Make sure you use the correct USERNAME for this forge/host. It might not be the same as on "github.com"!

Setting your Github Enterprise Username

For Github Enterprise instances you have to specify where the API can be accessed 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:

```
git config --global github.example.com/api/v3.user USERNAME
```

Make sure you use the correct USERNAME for this instance. It might not be the same as on "github.com"!

Doing this only tells Ghub who you are on this host, additionally you have to tell Ghub which repositories are connected to that forge/host, like so:

```
cd /path/to/repo
git config --local github.host example.com/api/v3
```

Setting your Username for Other Hosts and/or Forges

To inform Ghub about your username on HOST (a FORGE instance) do this:

```
git config --global FORGE.HOST.user USERNAME
```

FORGE can be one of `bitbucket`, `gitea` or `gogs`. It can also be `github` or `gitlab`; but if that is the case, then you should look at the preceding sections instead, which discuss these cases specifically.

HOST identifies the instance. This actually points at the top-level endpoint of the API and may contain path components, e.g.: `example.com/api`.

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

```
cd /path/to/repo
git config --local FORGE.HOST.user USERNAME
```

2.3 Creating and Storing a Token

2.3.1 Creating a Token

To create a token, use the web interface of the host you want to connect to. Here is a list of pages to do this for certain popular hosts:

- `https://github.com/settings/tokens`
- `https://gitlab.com/-/user_settings/personal_access_tokens`

For other forges we cannot provide a functioning URL because they contain unknown values such as your name. Just go to the general settings page of the respective host and then go from there.

Except on `gitea` and `gogs` each token can be limited to certain "scopes", i.e., it is possible to limit for which purposes any given token can be used.

Before you create a token to be used for a certain package, you should consult the documentation of that package, which in turn should tell you which scopes are needed and why. The Forge package for example does so in Section "Setup for Githubcom" in `forge` and Section "Setup for Gitlabcom" in `forge`.

2.3.2 Storing a Token

Please also see `auth` for all the gory details about Auth-Source.

The variable `auth-sources` controls how and where Auth-Source keeps its secrets. The default value is a list of three files: `("~/.authinfo" "~/.authinfo.gpg" "~/.netrc")`, but to avoid confusion you should make sure that only one of these files exists, and then you should also adjust the value of the variable to only ever use that file, for example:

```
(setq auth-sources '("~/.authinfo"))
```

In `~/.authinfo` secrets are stored in plain text. If you don't want that, then you should use the encrypted `~/.authinfo.gpg` instead:

```
(setq auth-sources '("~/.authinfo.gpg"))
```

Auth-Source also supports storing secrets in various external key-chains. See `auth` for more information.

The default 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$^{P}ACKAGE$" as the "login".

Assuming your **Github** username is "ziggy", the package is named "forge", and you want to access **Github.com** with the **token** "012345abcdef...", an entry in one of the three mentioned files would then look like this:

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

Assuming your **Gitlab** username is "ziggy", the package is named "forge", and you want to access **Gitlab.com** with the **token** "012345abcdef...", an entry in one of the three mentioned files would then look like this:

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

2.4 Github 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.)

3 API

3.1 Their APIs

Of course this manual does not cover the APIs of all forges that it supports, but for your convenience, here are the links to their API manuals:

- Github:
  - `https://docs.github.com/en/graphql` (GraphQl)
  - `https://docs.github.com/en/rest` (REST)
- Gitlab:
  - `https://docs.gitlab.com/api`
- Gitea:
  - `https://docs.gitea.com/next/development/api-usage`
  - `https://demo.gitea.com`
- Gogs:
  - `https://gogs.io/docs`
- Bitbucket:
  - `https://developer.atlassian.com/cloud/bitbucket`

3.2 Making REST Requests

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

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 headers are 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 depending 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 makes 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 package 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.
- 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 no error occurred, then it calls `CALLBACK`, unless that is `nil`.

  If an error occurred, then it calls `ERRORBACK`, or if that is `nil`, then `CALLBACK`. `ERRORBACK` can also be `t`, in which case it signals instead. `NOERROR` is ignored for all asynchronous requests.

  Both callbacks are called with four arguments.

  1. For `CALLBACK`, the combined value of the retrieved pages. For `ERRORBACK`, the error that occurred when retrieving the last page.

2. The headers of the last page as an alist.
3. Status information provided by `url-retrieve`. Its `:error` property holds the same information as the first argument to `ERRORBACK`.
4. 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 forth 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`.

3.3 Making GraphQL Requests

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

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 `GRAPHQL` 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 unpaginating 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 have implemented 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* &key *narrow username auth host forge* [Function]

This function is an opinionated alternative to `ghub-graphql`. It relies on 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 bundled `gsexp` library is used to turn that into a GraphQL query string. 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. If `CALLBACK` is `nil`, pretty-print the response.

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 *username auth host forge* [Function]

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 *username auth host forge* [Function]

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 *username auth host forge* [Function]

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 uses `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.

3.4 Github Convenience Wrappers

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

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

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* *unpaginate noerror reader username auth host callback errorback* [Function]

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

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

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

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 a timeout is reached. 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.

4 Notes

4.1 Using Ghub in Personal Scripts

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

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

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

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

When the `AUTH` argument is not specified, then a request is made on behalf of the `ghub` package itself. Like for any other package you have to create a dedicated token of course.

4.2 Using Ghub in Your Own 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.

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)
```

Keep in mind that the users of your package will have to manually create a suitable token. To make that easier, you should not only link to this manual but also prominently mention the scopes the token needs; and explain what they are needed for.

4.3 Forge Limitations and Notes

- There are no default Gitea and Gogs instances so 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 and my willingness to deal with that is extremely limited unless someone pays me vast amounts of money.
- 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.:

```
(ghub-get "/projects/python-mode-devs%2Fpython-mode"
          nil :forge 'gitlab)
```

Appendix A Function Index

Appendix B Variable Index