Quick Start Guide to gitlabr

Jirka Lewandowski jirka.lewandowski@wzb.eu

2017-04-24

Contents

1 Quick Start Example

R code using gitlabr to perform some easy, common gitlab actions can look like this:

```
library(gitlabr)

# connect as a fixed user to a gitlab instance
my_gitlab <- gl_connection("https://gitlab.points-of-interest.cc",
                           login = "testibaer",
                           password = readLines("secrets/gitlab_password.txt"))
# a function is returned
# its first argument is the request (name or function), optionally followed by parameters

my_gitlab(gl_list_projects)
```

```
## # A tibble: 2 × 36
##      id description default_branch public archived visibility_level
##   <chr>       <chr>          <chr>  <chr>    <chr>            <chr>
## 1    21                     master  FALSE    FALSE               10
## 2    20                     master   TRUE    FALSE               20
## # ... with 30 more variables: ssh_url_to_repo <chr>,
## #   http_url_to_repo <chr>, web_url <chr>, name <chr>,
## #   name_with_namespace <chr>, path <chr>, path_with_namespace <chr>,
## #   issues_enabled <chr>, merge_requests_enabled <chr>,
## #   wiki_enabled <chr>, builds_enabled <chr>, snippets_enabled <chr>,
## #   created_at <chr>, last_activity_at <chr>,
## #   shared_runners_enabled <chr>, lfs_enabled <chr>, creator_id <chr>,
## #   namespace.id <chr>, namespace.name <chr>, namespace.path <chr>,
## #   namespace.kind <chr>, namespace.full_path <chr>, star_count <chr>,
## #   forks_count <chr>, open_issues_count <chr>, public_builds <chr>,
## #   only_allow_merge_if_build_succeeds <chr>,
## #   request_access_enabled <chr>,
## #   permissions.project_access.access_level <chr>,
## #   permissions.project_access.notification_level <chr>
```

```r
my_gitlab(gl_list_files, project = "gitlabr", path = "R")
```

```
## # A tibble: 12 × 5
##                                          id                  name  type
##                                       <chr>                 <chr> <chr>
## 1  62975c0fa83178b215316e56e0b34798b2710f7a                  ci.R  blob
## 2  ca824ec1910fd72c38b9d16ec165b0c983347c7f            comments.R  blob
## 3  45070cc8bb0e788e0425b8edd2f2df19e2844ec7             connect.R  blob
## 4  c07004f7164fa6c5fe165fa937c1acc83ef668d5          gitlab_api.R  blob
## 5  59041ad608e39148bfe45dde25809cb4092daf07     gitlabr-package.R  blob
## 6  e1574cd4ce3c645882df44cb4c5c65779e74a51c          global_env.R  blob
## 7  83bfa9cdce966662574c3460b6d397d438c460f0              issues.R  blob
## 8  99b2436724f6721c8a0dc72fb128348921804db2      legacy_headers.R  blob
## 9  ecddbcec421d66be994f3034711279b8d496d3da magrittr_extensions.R  blob
## 10 fb3b9af4fe0a911f7b7f47855af09dde1011a740  projects_and_repos.R  blob
## 11 a5364535814102af5d1c50c86519c16c2668ef2e  shiny_module_login.R  blob
## 12 eced853c61a2a1a138bc457a412a9fc0ccdce97d         update_code.R  blob
## # ... with 2 more variables: path <chr>, mode <chr>
```

```r
# create a new issue
new_feature_issue <- my_gitlab(gl_new_issue, project = "testor", "Implement new feature")

# requests via gitlabr always return data_frames, so you can use all common manipulations
require(dplyr)
example_user <-
  my_gitlab("users") %>%
    filter(username == "testibaer")

# assign issue to a user
my_gitlab(gl_assign_issue, project = "testor",
          new_feature_issue$iid,
          assignee_id = example_user$id)
```

```
## # A tibble: 1 × 25
##      id   iid project_id                 title  state
##   <chr> <chr>      <chr>                 <chr>  <chr>
## 1   224    41         21 Implement new feature opened
## # ... with 20 more variables: created_at <chr>, updated_at <chr>,
## #   assignee.name <chr>, assignee.username <chr>, assignee.id <chr>,
## #   assignee.state <chr>, assignee.avatar_url <chr>,
## #   assignee.web_url <chr>, author.name <chr>, author.username <chr>,
## #   author.id <chr>, author.state <chr>, author.avatar_url <chr>,
## #   author.web_url <chr>, user_notes_count <chr>, upvotes <chr>,
## #   downvotes <chr>, confidential <chr>, web_url <chr>, subscribed <chr>
```

```r
my_gitlab(gl_list_issues, "testor", state = "opened")
```

```
## # A tibble: 2 × 26
##      id   iid project_id                 title  state
##   <chr> <chr>      <chr>                 <chr>  <chr>
## 1   224    41         21 Implement new feature opened
## 2    83     1         21            test issue opened
## # ... with 21 more variables: created_at <chr>, updated_at <chr>,
## #   assignee.name <chr>, assignee.username <chr>, assignee.id <chr>,
## #   assignee.state <chr>, assignee.avatar_url <chr>,
## #   assignee.web_url <chr>, author.name <chr>, author.username <chr>,
## #   author.id <chr>, author.state <chr>, author.avatar_url <chr>,
## #   author.web_url <chr>, user_notes_count <chr>, upvotes <chr>,
## #   downvotes <chr>, confidential <chr>, web_url <chr>, subscribed <chr>,
## #   description <chr>
```

```r
# close issue
my_gitlab(gl_close_issue, project = "testor", new_feature_issue$iid)$state
```

```
## [1] "closed"
```

2 Central features of gitlabr

- `gitlabr` provides a high and a low level interface to the gitlab API at the same time:
  - Common queries are wrapped in special convenience functions that can be used without any knowledge of the gitlab API itself – find a list to start right away in the section "Convenience function list".
  - Still, the package can be used to access the complete gitlab API – learn how to use its full power in the section "API calls".
- The output of every call to a `gitlabr` function is a `data_frame` to integrate seamless into dplyr's data manipulation mindset
- Pagination is wrapped for the user, but can be controlled via parameters `page` and `per_page` if necessary.
- To allow programming in your favorite style, everything you can do with `gitlabr` you can do using any of a set of general idioms – get to know them in the section "Different ways to do it".
- You can write your own convenience wrappers on top of the `gitlabr` logic following only one principle as described in the section "Writing custom gitlab request functions".

3 API calls

This section describes how R function calls are translated into HTTP requests to the gitlab API (`gitlabr`'s "low level interface"). For a documentation using `gitlabr` without knowledge of the gitlab API (`gitlabr`'s "high level interface"), see the "Quick Start Example" above, the "Convenience function list" below or the individual function documentation.

The core function of the low level interface is `gitlab`, with the help of which arbitrary calls to the gitlab API can be formulated. It takes as required arguments the request location as a character vector, API endpoint URL and HTTP verb and passes additional arguments as query parameters (keeping their names) on to the API request.

```r
gitlab(c("projects", 12, "issues"),
       api_root = "https://gitlab.points-of-interest.cc/api/v3",
       private_token = "XXX", # authentication for API
       verb = httr::GET,  # defaults to GET, but POST, PUT, DELETE can be used likewise
       state = "active") # additional parameters (...) for the query
```

translates to

```
GET https://gitlab.points-of-interest.cc/api/v4/projects/12/issues?state=active&private_token=XXX
```

This way, any request documented in the Gitlab API documentation can be issued from `gitlabr`.

The high level interface consists of a number of functions that each have additional arguments from which the request location is constructed, while all other arguments are simply passed on to `gitlab`. For example:

```r
gl_edit_issue(project = "test-project", 12, description = "Really cool new feature",
           api_root = "...", private_token = "XXX")
```

does nothing but

```r
gitlab(c("projects",
         4,  # numeric id of test-project is found by search
         "issues",
         12),
       description = "Really cool new feature",
       api_root = "...",
       private_token = "XXX",
       verb = httr::PUT))
```

and hence translates to

```
PUT .../projects/4/issues/12?private_token=XXX?description=Really%20cool%20new%20feature
```

To spare you the repetitive task of specifying the API root and key in every call, you can use `gitlab_connection` as described in the next section "Different ways to do it".

Note: currently (gitlabr version 0.9) gitlab API v4 is supported. Support for Gitlab API v3 (from Gitlab version < 9.0) is still included via flag parameters, but will be deprecated from gitlab version 1.0 onwards. For details see the section "API version" of the documentation of `gl_connection`.

4 Different ways to do it

`gitlabr` is implemented following the functional programming paradigm. Several of its functions return or accept functions as arguments. This results in huge flexibility in how API requests using `gitlabr` can be formulated your R code. Three major styles are described below, after introducing the central mechanism of creating more specific API connection functions.

4.1 Creating connections

The idea of connections in `gitlabr` is to generate functions with the same signature and capability of the central API call function `gitlab`, but with certain parameters set to fixed values ("curried"). This way these more specialized functions represent and provide the connection – for example – to a specific gitlab instance as a specific user. Such specialized functions can be created by the function `gitlab_connection` and then used exactly as you would use `gitlab`:

```r
my_gitlab <- gl_connection("https://gitlab.points-of-interest.cc/",
                           login = "jlewando",
                           password = readLines("secrets/gitlab_password.txt"))
my_gitlab("projects")
```

```
## # A tibble: 1 × 36
##      name    id description default_branch public archived
##     <chr> <chr>       <chr>          <chr>  <chr>    <chr>
## 1 gitlabr    20                     master   TRUE    FALSE
## # ... with 30 more variables: visibility_level <chr>,
## #   ssh_url_to_repo <chr>, http_url_to_repo <chr>, web_url <chr>,
## #   name_with_namespace <chr>, path <chr>, path_with_namespace <chr>,
## #   issues_enabled <chr>, merge_requests_enabled <chr>,
## #   wiki_enabled <chr>, builds_enabled <chr>, snippets_enabled <chr>,
## #   created_at <chr>, last_activity_at <chr>,
## #   shared_runners_enabled <chr>, lfs_enabled <chr>, creator_id <chr>,
## #   namespace.id <chr>, namespace.name <chr>, namespace.path <chr>,
## #   namespace.kind <chr>, namespace.full_path <chr>, star_count <chr>,
## #   forks_count <chr>, open_issues_count <chr>, public_builds <chr>,
## #   only_allow_merge_if_build_succeeds <chr>,
## #   request_access_enabled <chr>,
## #   permissions.project_access.access_level <chr>,
## #   permissions.project_access.notification_level <chr>
```

`gitlab_connection` can take arbitrary parameters, returning a function that issues API requests with these parameter values set. In addition, the different ways to login in the gitlab API (see https://doc.gitlab.com/ce/api/session.html) are all auto-detected and handled by replacing the login information with a private authentification token provided by the API in future calls using the returned function.

As a convenience wrapper to directly connect to a specific project in a gitlab instance, `project_connection` exists.

For combining so created gitlab connections with the convenience functions to perform common tasks, several possible styles/idioms exist:

4.2 function-in-function style

Instead of the query as character vector `gitlab` and thus also all connections accept equivalently a *function* as first argument, that is then called with the additional parameters and using the connection for all API calls:

```r
my_gitlab(gl_new_issue, "Implement new feature", project = "testor")
```

`new_issue` is an example function here, the principle style works for all convenience functions of `gitlabr` listed in the "Convenience function list" below or user-defined as described in the section "Writing custom gitlab request functions".

Some of the convenience perform additional transformation or renaming of parameters. Hence, the parameters given to the exemplary `my_gitlab(...)` call after the function should be valid according the documentation of the respective function and may differ from names used in the gitlab API itself, although this is the case only in very few cases.

4.3 parameter style

Alternatively, `gitlab` as well as all convenience wrappers accept a parameter `gitlab_con` specifying the function to use for the actual API call. Hence, you can pass a gitlab connection (as returned by `gitlab_connection`) with the R function call:

```r
gl_new_issue("Implement new feature", project = "testor", gitlab_con = my_gitlab)
```

Again, `new_issue` is an example function here, the principle style works for all convenience functions of `gitlabr` listed in the "Convenience function list" below or user-defined as described in the section "Writing custom gitlab request functions".

4.4 set style

In order to avoid the repeated specification of `gitlab_con` in the parameter style, you can also set a global variable managed by gitlabr to use a specific connection function for every call:

```r
set_gitlab_connection(my_gitlab)
gl_new_issue("Implement new feature", project = "testor")
```

Again, `new_issue` is an example function here, the principle style works for all convenience functions of `gitlabr` listed in the "Convenience function list" below or user-defined as described in the section "Writing custom gitlab request functions".

Note that the set style is not purely functional, since `set_gitlab_connection` changes a saved global variable affecting the results of all future `gitlab` calls. You can reset this variable to the default value using `unset_gitlab_connection()`.

5 Convenience function list

Here is a list of all convenience wrapper functions that come with `gitlabr` 0.9 Only function names are given, since they are designed to be self-explanatory. For reference on how to use each function, refer to its R documentation using the `?` operator.

- gl_assign_issue
- gl_builds
- gl_close_issue
- gl_create_branch
- gl_create_merge_request
- gl_comment_commit
- gl_comment_issue
- gl_delete_branch
- gl_edit_comment
- gl_edit_commit_comment
- gl_edit_issue
- gl_edit_issue_comment
- gl_file_exists
- gl_get_comments
- gl_get_commit_comments
- gl_get_commits
- gl_get_diff
- gl_get_file
- gl_get_issue
- gl_get_issue_comments
- gl_jobs
- gl_latest_build_artifact
- gl_list_issues
- gl_list_branches
- gl_list_files
- gl_list_projects
- gl_new_issue
- gl_pipelines
- gl_push_file
- gl_reopen_issue
- gl_repository
- gl_unassign_issue

Note: There are more locations and actions that can be accessed through the gitlab API. See the documentation of the Gitlab API for this. The next section describes how you can write your own convenience wrappers that work with all styles described in the section "Different ways to do it".

Note also that since gitlabr version 0.7 all functions of this package follow a consistent naming scheme, starting with `gl_`. The old function names are deprecated and will be removed with gitlabr version 1.0.

6 Writing custom gitlab request functions

It is very easy to write your own convenience wrappers for accessing API endpoints you whish and make sure they fully integrate into `gitlabr` and work conveniently with all connection and call idioms described in this vignette. The only requirement to your function is that it executes an R function call to `gitlab` (or another convenience function) to which the `...` argument is passed on.

That is, a simple function to block users directly from R is as simple as:

```r
block_user <- function(uid, ...) {
  gitlab(c("users", uid, "block"),  ## for API side documentation see:
         verb = httr::PUT, ## https://doc.gitlab.com/ce/api/users.html#block-user
         ...) ## don't forget the dots to make gitlabr features fully available
}
```

More hints for more convenience:

- To be consistent with another important `gitlabr` principle, make sure your function returns a `data_frame` (which it does if you simply pass up the return value of `gitlab` or one of the package's own convenience functions). `gitlab` has some heuristics to format the API response to a `data_frame`, if these fail for your specific request, you can pass `auto_format = FALSE` and format the response manually.
- To translate project names to numeric ids automatically, you can use `gitlabr`'s internal functions `proj_req` translating the request location.
- To translate user-visible project-wide issue ids to global ids used by the gitlab API, you can use `gitlabr`'s internal function `to_issue_id` when constructing the request.

And last but not least, if you've written a convenience wrapper for yourself, keep in mind that it might be of help to many others and you can contribute it to `gitlabr` on https://gitlab.points-of-interest.cc/points-of-interest/gitlabr/.

7 Using gitlab CI with gitlabr

`gitlabr` can also be used to create a `.gitlab-ci.yml` file to test, build and check an R package using gitlabs CI software. See the `use_gitlab_ci` and related functions for documentation.