

Quick Start Guide to gitlabr

Jirka Lewandowski jirka.lewandowski@wzb.eu

2015-10-25

Contents

1	Quick Start Example	1
2	Central features of gitlabr	3
3	API calls	3
4	Different ways to do it	4
4.1	Creating connections	4
4.2	function-in-function style	5
4.3	parameter style	6
4.4	set style	6
5	Convenience function list	6
6	Writing custom gitlab request functions	7

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 <- gitlab_connection("https://gitlab.points-of-interest.cc",
                              login = "jlewando",
                              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(list_projects)

## Source: local data frame [7 x 35]
##
##       name      id      description default_branch public
##       (chr) (chr)      (chr)      (chr)      (chr)
## 1 packagetrackr  24      master      TRUE
## 2 webpage       22      master      TRUE
## 3 gitlabr       20      master      TRUE
```

```
## 4      arg      18 arg software from BSc thesis      master  TRUE
## 5      topology  10                                master  TRUE
## 6 QCA sensitivity 6                                master  TRUE
## 7      QCAtools  5                                master  TRUE
## Variables not shown: archived (chr), visibility_level (chr),
## ssh_url_to_repo (chr), http_url_to_repo (chr), web_url (chr), owner.name
## (chr), owner.username (chr), owner.id (chr), owner.state (chr),
## owner.avatar_url (chr), owner.web_url (chr), name_with_namespace (chr),
## path (chr), path_with_namespace (chr), issues_enabled (chr),
## merge_requests_enabled (chr), wiki_enabled (chr), snippets_enabled
## (chr), created_at (chr), last_activity_at (chr), creator_id (chr),
## namespace.id (chr), namespace.name (chr), namespace.path (chr),
## namespace.owner_id (chr), namespace.created_at (chr),
## namespace.updated_at (chr), namespace.description (chr), star_count
## (chr), forks_count (chr)
```

```
my_gitlab(list_files, project = "gitlabr", path = "R")
```

```
## Source: local data frame [8 x 4]
##
##              id              name type
##              (chr)          (chr) (chr)
## 1 ff8af56dd43ddaedab7c5f6598f26386823e6bed      comments.R blob
## 2 d905389741b59e12622d4d5ff6e224beb3eb9e54      connect.R blob
## 3 286f11fa48694bd544a892164b04be3e60c627a7      gitlab_api.R blob
## 4 9df46bcf99ae79973bbffbe9a463dd078f77443c      gitlabr-package.R blob
## 5 7fd0473f969aced0c07be51b3d1f820446787a53      global_env.R blob
## 6 ea18bfd9fb6a9026ef14bfc96013f2fdafdaf8b83      issues.R blob
## 7 2ad7f08fd7bbae47e5adabda0e40a77a1dcfe97d      magrittr_extensions.R blob
## 8 3b7c311db184b55cd1289cdf97c3c7438245234      projects_and_repos.R blob
## Variables not shown: mode (chr)
```

```
# create a new issue
new_feature_issue <- my_gitlab(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(assign_issue, project = "testor",
  new_feature_issue$id,
  assignee_id = example_user$id)
```

```
## Source: local data frame [1 x 19]
##
##      id  iid project_id      title  state
##   (chr) (chr)    (chr)      (chr) (chr)
## 1  141   26       21 Implement new feature opened
## Variables not shown: 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)

my_gitlab(get_issues, "testor", state = "opened")

## Source: local data frame [2 x 20]
##
##      id   iid project_id          title  state
##   (chr) (chr)   (chr)      (chr)   (chr)
## 1   141   26       21 Implement new feature opened
## 2    83    1       21      test issue opened
## Variables not shown: 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), description (chr)

# close issue
my_gitlab(close_issue, project = "testor", new_feature_issue$id)$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 R’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.

```
gitlab(c("projects", 12, "issues"),
      api_root = "https://gitlab.points-of-interest.cc/api/v3",
      private_token = "XXX", # authentication for API
      verb = http::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/v3/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:

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

does nothing but

```
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 = http::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](#)”.

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`:

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

```
## Source: local data frame [7 x 35]
##
##           name      id      description default_branch public
##           (chr) (chr)      (chr)          (chr)      (chr)
## 1 packagetrackr    24
## 2      webpage    22
## 3      gitlabr    20
## 4      arg      18 arg software from BSc thesis
## 5      topology   10
## 6 QCA sensitivity    6
## 7      QCAtools    5
## Variables not shown: archived (chr), visibility_level (chr),
## ssh_url_to_repo (chr), http_url_to_repo (chr), web_url (chr), owner.name
## (chr), owner.username (chr), owner.id (chr), owner.state (chr),
## owner.avatar_url (chr), owner.web_url (chr), name_with_namespace (chr),
## path (chr), path_with_namespace (chr), issues_enabled (chr),
## merge_requests_enabled (chr), wiki_enabled (chr), snippets_enabled
## (chr), created_at (chr), last_activity_at (chr), creator_id (chr),
## namespace.id (chr), namespace.name (chr), namespace.path (chr),
## namespace.owner_id (chr), namespace.created_at (chr),
## namespace.updated_at (chr), namespace.description (chr), star_count
## (chr), forks_count (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 <http://doc.gitlab.com/ce/api/session.html>) are all auto-detected and handled by replacing the login information with a private authentication 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:

```
my_gitlab(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:

```
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:

```
set_gitlab_connection(my_gitlab)
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` 0.5. 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.

- `assign_issue`
- `close_issue`
- `comment_commit`
- `comment_issue`
- `comments`
- `compare_refs`
- `edit_comment`
- `edit_commit_comment`
- `edit_issue`
- `edit_issue_comment`
- `get_comments`
- `get_commit_comments`
- `get_commits`
- `get_diff`
- `get_file`
- `get_issue`

- `get_issue_comments`
- `get_issues`
- `list_files`
- `list_projects`
- `new_issue`
- `reopen_issue`
- `repository`
- `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](#)”.

6 Writing custom gitlab request functions

It is very easy to write your own convenience wrappers for accessing API endpoints you wish 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:

```
block_user <- function(uid, ...) {
  gitlab(c("users", uid, "block"), ## for API side documentation see:
    verb = httr::PUT, ## http://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 <http://gitlab.points-of-interest.cc/points-of-interest/gitlabr/>.