

Package ‘ghclass’

August 31, 2026

Title Tools for Managing Classes on GitHub

Version 0.4.2

Description Interface for the GitHub API that enables efficient management of courses on GitHub. It has a functionality for managing organizations, teams, repositories, and users on GitHub and helps automate most of the tedious and repetitive tasks around creating and distributing assignments.

License GPL-3

URL <https://github.com/rundel/ghclass>,
<https://rundel.github.io/ghclass/>

BugReports <https://github.com/rundel/ghclass/issues>

Depends R (>= 3.4.0)

Imports base64enc, fs, gh (>= 1.4.0), glue, httr, lubridate, purrr,
rlang, stringr, tibble, whisker, withr, dplyr, cli (>= 3.0.0),
lifecycle, readr, tidyselect

Suggests here, knitr, rmarkdown, sodium, styler, testthat (>= 3.0.0),
usethis, gert, gitcreds

Encoding UTF-8

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

NeedsCompilation no

Author Colin Rundel [aut, cre],
Mine Cetinkaya-Rundel [aut],
Therese Anders [ctb]

Maintainer Colin Rundel <rundel@gmail.com>

Repository CRAN

Date/Publication 2026-08-31 16:20:09 UTC

Contents

action	2
action_badge	5
branch	6
github_api_limit	7
github_orgs	9
github_rate_limit	9
github_whoami	11
github_with_pat	12
issue	13
local_repo	15
local_repo_anonymize	17
local_repo_rename	19
org_create_assignment	20
org_details	21
org_grade_assignment	23
org_members	24
org_perm	26
pages	27
pr	28
repo_core	29
repo_details	32
repo_file	35
repo_notification	37
repo_style	38
repo_user	40
team	42
team_members	43
team_roster	45
user	46
Index	48

action	<i>Retrieve information about GitHub Actions workflows and their runs.</i>
--------	--

Description

- action_workflows() - retrieve details on repo workflows.
- action_runs() - retrieve details on repo workflow runs.
- action_status() - DEPRECATED - retrieve details on most recent workflow runs.
- action_runtime() - retrieves runtime durations for workflow runs.
- action_artifacts() - retrieve details on available workflow artifacts.
- action_artifact_download() - downloads artifact(s) into a local directory.
- action_artifact_delete() - deletes artifact(s).
- action_run_logs() - downloads workflow run log(s) into a local directory.

Usage

```
action_artifacts(  
  repo,  
  filter = NULL,  
  exclude = FALSE,  
  keep_expired = FALSE,  
  which = c("latest", "all"),  
  filter_branch = NULL,  
  exclude_branch = FALSE  
)  
  
action_artifact_delete(repo, ids)  
  
action_artifact_download(  
  repo,  
  dir,  
  filter = NULL,  
  exclude = FALSE,  
  filter_branch = NULL,  
  exclude_branch = FALSE,  
  keep_zip = FALSE,  
  nest = FALSE,  
  overwrite = FALSE,  
  ids = action_artifacts(repo, filter = filter, exclude = exclude, filter_branch =  
    filter_branch, exclude_branch = exclude_branch)  
)  
  
action_run_logs(  
  repo,  
  dir,  
  run_ids = action_runs(repo),  
  keep_zip = FALSE,  
  overwrite = FALSE  
)  
  
action_runs(  
  repo,  
  branch = NULL,  
  event = NULL,  
  status = NULL,  
  created = NULL,  
  limit = 1  
)  
  
action_status(  
  repo,  
  branch = NULL,  
  event = NULL,
```

```

    status = NULL,
    created = NULL,
    limit = 1
)

action_runtime(
  repo,
  branch = NULL,
  event = NULL,
  status = NULL,
  created = NULL,
  limit = 1
)

action_workflows(repo, full = FALSE)

```

Arguments

repo	Character. Address of repository in owner/name format.
filter	Character. Regex pattern to filter artifacts by artifact name.
exclude	Logical. If TRUE, exclude artifact names matching filter instead of including them. Default FALSE.
keep_expired	Logical. Should expired artifacts be returned. Default FALSE.
which	Character. Either "latest" to return only the most recent of each artifact or "all" to return all artifacts.
filter_branch	Character. Regex pattern to filter artifacts by branch name.
exclude_branch	Logical. If TRUE, exclude branches matching filter_branch instead of including them. Default FALSE.
ids	Integer or data frame. Artifact ids to be downloaded or deleted. If a data frame is passed then the id column will be used.
dir	Character. Path to the directory where logs or artifacts will be saved.
keep_zip	Logical. Should the log zips be saved (TRUE) or their contents extracted (FALSE). Default FALSE.
nest	Logical. If TRUE, place each repo's artifacts under a dir/{repo_name}/ subdirectory; otherwise flatten into dir/ with repo-prefixed names. Default FALSE.
overwrite	Logical. Should existing files be overwritten. Default FALSE.
run_ids	Integer or data frame. Run ids for which to download logs. If a data frame is passed then the run_id column will be used. Defaults to the most recent run for each repo (i.e. action_runs(repo)).
branch	Character. Filter runs associated with a particular branch.
event	Character. Filter runs for triggered by a specific event. See here for possible event names.
status	Character. Filter runs for a particular status or conclusion (e.g. completed or success).

created	Character. Filter runs for a given creation date. See here for date query syntax.
limit	Numeric. Maximum number of workflow runs to return. Default 1. Note results are chronologically ordered, so <code>limit = 1</code> will return the most recent action run for a repository.
full	Logical. Should all workflow columns be returned. Default FALSE.

Value

`action_workflows()`, `action_runs()`, `action_runtime()`, and `action_artifacts` all return tibbles containing information on requested repos' available workflows, recent workflow runs, workflow runs runtimes, and generated artifacts respectively.

`action_artifact_download()` returns an invisible character vector containing the paths of the extraction directories for each successfully downloaded artifact (NA entries indicate failures).

`action_artifact_delete()` returns an invisible data frame containing repository names and ids of the deleted artifacts.

`action_run_logs()` returns an invisible character vector containing the paths of all downloaded log files or directories.

Examples

```
## Not run:
action_workflows("rundel/ghclass")

action_runs("rundel/ghclass")

action_runtime(c("rundel/ghclass", "rundel/parsermd"))

action_artifacts(c("rundel/ghclass", "rundel/parsermd"))

## End(Not run)
```

action_badge

Add or remove GitHub Actions badges from a repository

Description

- `action_add_badge()` - Add a GitHub Actions badge to a file.
- `action_remove_badge()` - Remove one or more GitHub Action badges from a file.

Usage

```
action_add_badge(
  repo,
  workflow = NULL,
  where = "^.",
```

```

    line_padding = "\n\n\n",
    file = "README.md"
)

action_remove_badge(repo, workflow_pat = ".*?", file = "README.md")

```

Arguments

repo	Character. Address of repository in owner/name format.
workflow	Character. Name of the workflow.
where	Character. Regex pattern indicating where to insert the badge, defaults to the beginning of the target file.
line_padding	Character. What text should be added after the badge.
file	Character. Target file to be modified, defaults to README.md#'
workflow_pat	Character. Name of the workflow to be removed, or a regex pattern that matches the workflow name.

Value

Both `action_add_badge()` and `action_remove_badge()` invisibly return a list containing the results of the relevant GitHub API call.

Note

`action_add_badge()` is not idempotent; re-running it adds another badge block, so use `action_remove_badge()` first if you need to avoid duplicates.

Examples

```

## Not run:
action_add_badge("ghclass-test/hw1-team01")

action_remove_badge("ghclass-test/hw1-team01")

## End(Not run)

```

branch

Create and delete branches in a repository

Description

- `branch_create()` - creates a new branch from an existing GitHub repo.
- `branch_delete()` - deletes a branch from an existing GitHub repo.
- `branch_remove()` - previous name of `branch_delete`, deprecated.

Usage

```
branch_create(repo, branch, new_branch)
```

```
branch_delete(repo, branch)
```

```
branch_remove(repo, branch)
```

Arguments

repo	GitHub repository address in owner/repo format.
branch	Repository branch to use.
new_branch	Name of branch to create.

Value

branch_create() and branch_delete() invisibly return a list containing the results of the relevant GitHub API call.

See Also

[repo_branches](#)

Examples

```
## Not run:
repo_create("ghclass-test", "test_branch", auto_init=TRUE)

branch_create("ghclass-test/test_branch", branch = "main", new_branch = "test")
repo_branches("ghclass-test/test_branch")

branch_delete("ghclass-test/test_branch", branch="test")
repo_branches("ghclass-test/test_branch")

repo_delete("ghclass-test/test_branch", prompt = FALSE)

## End(Not run)
```

github_api_limit

Tools for limiting gh's GitHub api requests.

Description

- github_get_api_limit() - returns the current limit on results returned by gh.
- github_set_api_limit() - sets a limit on results returned by gh.
- github_get_max_wait() / github_set_max_wait() - get or set the maximum number of seconds gh will wait when it hits a rate limit. NULL (the default) uses gh's own default.

- `github_get_max_rate()` / `github_set_max_rate()` - get or set the maximum request rate, in requests per second, used to proactively throttle requests and avoid secondary rate limits. NULL (the default) disables throttling.

Usage

```
github_get_api_limit()

github_set_api_limit(limit = 10000L)

github_get_max_wait()

github_set_max_wait(max_wait = NULL)

github_get_max_rate()

github_set_max_rate(max_rate = NULL)
```

Arguments

<code>limit</code>	The maximum number of records to return from an API request.
<code>max_wait</code>	The maximum number of seconds to wait when rate limited, or NULL to use gh's default. Passed to <code>gh::gh()</code> 's <code>.max_wait</code> .
<code>max_rate</code>	The maximum request rate in requests per second, or NULL for no throttling. Passed to <code>gh::gh()</code> 's <code>.max_rate</code> .

Details

These values are stored in the `"ghclass.api.limit"`, `"ghclass.max.wait"`, and `"ghclass.max.rate"` options globally.

Value

`github_get_api_limit()` returns a single integer value.

`github_get_max_wait()` and `github_get_max_rate()` return their stored value or NULL.

`github_set_*`() functions invisibly return the value of their argument.

Examples

```
github_get_api_limit()

github_set_api_limit(500)

github_get_api_limit()
```

github_orgs	<i>Collect details on the authenticated user's GitHub organization memberships (based on the current PAT).</i>
-------------	--

Description

Collect details on the authenticated user's GitHub organization memberships (based on the current PAT).

Usage

```
github_orgs(quiet = FALSE)
```

Arguments

quiet Logical. Should status messages be shown. Default FALSE.

Value

Returns a tibble with organization details.

Examples

```
## Not run:  
github_orgs()  
  
## End(Not run)
```

github_rate_limit	<i>Tools for handling GitHub personal access tokens (PAT)</i>
-------------------	---

Description

- `github_get_token` - returns the user's GitHub personal access token (PAT).
- `github_set_token` - defines the user's GitHub PAT by setting the `GITHUB_PAT` environmental variable. This value will persist until the session ends or `github_reset_token()` is called.
- `github_reset_token` - removes the value stored in the `GITHUB_PAT` environmental variable.
- `github_test_token` - checks if a PAT is valid by attempting to authenticate with the GitHub API.
- `github_token_scopes` - returns a vector of scopes granted to the token.
- `github_token_sitrep` - reports on the token: its type and source, the authenticated user, the granted scopes (flagging any that ghclass needs but are missing), and the API rate limit.

Usage

```
github_rate_limit()

github_graphql_rate_limit()

github_get_token()

github_set_token(token)

github_reset_token()

github_test_token(token = github_get_token())

github_token_scopes(token = github_get_token())

github_token_sitrep(token = github_get_token())
```

Arguments

token Character. Either the literal token, or the path to a file containing the token.

Details

This package looks for the personal access token (PAT) in the following places (in order):

- Value of GITHUB_PAT environmental variable.
- A credential returned by `gh::gh_token()`, such as GITHUB_TOKEN, a token stored with `gitcreds`, or a Posit Connect viewer token.

For additional details on creating a GitHub PAT see the `usethis` vignette on [Managing Git\(Hub\) Credentials](#). For those who do not wish to read the entire article, the quick start method is to use:

- `usethis::create_github_token()` - to create the token and then,
- `gitcreds::gitcreds_set()` - to securely cache the token.

Scopes:

A classic PAT needs the `repo` and `admin:org` scopes for `ghclass` to manage an organization's repositories and teams, the `workflow` scope to add or modify files under `.github/workflows/`, the `notifications` scope to use `repo_watch()`, `repo_ignore()`, and `repo_unwatch()`, and the `delete_repo` scope to use `repo_delete()`. Note that `usethis::create_github_token()` does not select `admin:org` by default. Fine-grained tokens do not report scopes, so `github_token_sitrep()` cannot check them. GitHub does not support fine-grained or GitHub App tokens for `repo_watch()`, `repo_ignore()`, or `repo_unwatch()`.

Value

`github_get_token()` returns the current PAT as a character string with the `gh_pat` class. See [gh::gh_token\(\)](#) for additional details.

github_set_token() and github_reset_token() return the result of Sys.setenv() and Sys.unsetenv() respectively.

github_test_token() invisibly returns a logical value, TRUE if the test passes, FALSE if not.

github_token_scopes() returns a character vector of granted scopes.

github_token_sitrep() invisibly returns a list with the token's type, source, API url, the authenticated user's login, the granted scopes (NULL when the token does not report them), the scopes ghclass uses that are missing, and rate limit details.

Examples

```
## Not run:
github_test_token()

github_token_scopes()

github_token_sitrep()

(pat = github_get_token())

github_set_token("ghp_BadTokenBadTokenBadTokenBadTokenBadToken")
github_get_token()
github_test_token()

github_set_token(pat)

## End(Not run)
```

github_whoami	<i>Returns the login of the authenticated user (based on the current PAT).</i>
---------------	--

Description

Returns the login of the authenticated user (based on the current PAT).

Usage

```
github_whoami(quiet = FALSE)
```

Arguments

quiet Logical. Should status messages be shown. Default FALSE.

Value

Character value containing user login.

Examples

```
## Not run:  
github_whoami()  
  
## End(Not run)
```

github_with_pat	<i>withr-like functions for temporary personal access token</i>
-----------------	---

Description

Temporarily change the GITHUB_PAT environmental variable for GitHub authentication. Based on the withr interface.

Usage

```
with_pat(new, code)  
  
local_pat(new, .local_envir = parent.frame())
```

Arguments

new	Temporary GitHub access token
code	Code to execute with the temporary token
.local_envir	The environment to use for scoping.

Details

if new = NA is used the GITHUB_PAT environment variable will be unset.

Value

The results of the evaluation of the code argument.

Examples

```
## Not run:  
with_pat("1234", print(github_get_token()))  
  
## End(Not run)
```

Description

- `issue_create` creates a new issue.
- `issue_close` closes an existing issue.
- `issue_edit` edits the properties of an existing issue.

Usage

```
issue_close(repo, number, comment = NULL)
```

```
issue_create(  
  repo,  
  title,  
  body,  
  labels = character(),  
  assignees = character(),  
  delay = 0  
)
```

```
issue_edit(  
  repo,  
  number,  
  title = NULL,  
  body = NULL,  
  state = NULL,  
  milestone = NULL,  
  labels = list(),  
  assignees = list()  
)
```

Arguments

<code>repo</code>	Character. Address of one or more repositories in owner/name format.
<code>number</code>	Integer. GitHub issue number.
<code>comment</code>	Character. Optional comment to post on the issue before closing it. If posting the comment fails the issue is not closed. Recycled against <code>repo / number</code> .
<code>title</code>	Character. Title of the issue.
<code>body</code>	Character. Content of the issue.
<code>labels</code>	Character. Vector of the labels to associate with this issue
<code>assignees</code>	Character. Vector of logins for users assigned to the issue.
<code>delay</code>	Numeric. Delay between each API request. Issue creation has a secondary rate limit (~ 20/min).

state	Character. State of the issue. Either "open" or "closed".
milestone	Character. The number of the milestone to associate this issue with. Only users with push access can set the milestone for issues. The milestone is silently dropped otherwise.

Value

All functions invisibly return a list containing the results of the relevant GitHub API call.

See Also

[repo_issues](#)

Examples

```
## Not run:
repo_create("ghclass-test","test_issue")

issue_create(
  "ghclass-test/test_issue",
  title = "Issue 1",
  body = "This is an issue"
)

issue_create(
  "ghclass-test/test_issue",
  title = "Issue 2", body = "This is also issue",
  label = "Important"
)

issue_create(
  "ghclass-test/test_issue",
  title = "Issue 3", body = "This is also issue",
  label = c("Important", "Super Important"),
  assignees = "rundel"
)

issue_close("ghclass-test/test_issue", 1)

issue_edit(
  "ghclass-test/test_issue", 2,
  title = "New issue 2 title!",
  body = "Replacement body text"
)

ghclass::repo_issues("ghclass-test/test_issue")

repo_delete("ghclass-test/test_issue", prompt=FALSE)

## End(Not run)
```

Description

- `local_repo_clone()` - Clones a GitHub repository to a local directory.
- `local_repo_add()` - Equivalent to `git add` - stages a file in a local repository.
- `local_repo_commit()` - Equivalent to `git commit` - commits staged files in a local repository.
- `local_repo_push()` - Equivalent to `git push` - push a local repository.
- `local_repo_pull()` - Equivalent to `git pull` - pull a local repository.
- `local_repo_branch()` - Equivalent to `git branch` - create a branch in a local repository.
- `local_repo_log()` - Equivalent to `git log` - returns a data frame for git log entries.

Usage

```
local_repo_add(repo_dir, files = ".")

local_repo_branch(repo_dir, branch)

local_repo_clone(
  repo,
  local_path = ".",
  branch = NULL,
  mirror = FALSE,
  verbose = FALSE
)

local_repo_commit(repo_dir, message)

local_repo_log(repo_dir, max = 100)

local_repo_pull(repo_dir, verbose = FALSE)

local_repo_push(
  repo_dir,
  remote = "origin",
  branch = NULL,
  force = FALSE,
  prompt = TRUE,
  mirror = FALSE,
  verbose = FALSE
)
```

Arguments

repo_dir	Vector of repo directories or a single directory containing one or more repos.
files	Files to be staged
branch	Repository branch to use.
repo	GitHub repo address with the form owner/name.
local_path	Local directory to store cloned repos.
mirror	Equivalent to --mirror
verbose	Display verbose output.
message	Commit message
max	Maximum number of log entries to retrieve per repo.
remote	Repository remote to use.
force	Force push?
prompt	Prompt before force push?

Details

All local_repo_* functions depend on the gert library being installed.

Value

local_repo_clone() invisibly returns a character vector of paths for the local repo directories.

local_repo_log() returns a tibble containing repository details.

All other functions invisibly return a list containing the results of the relevant call to gert.

Examples

```
## Not run:
repo = repo_create("ghclass-test", "local_repo_test")

dir = file.path(tempdir(), "repos")
local_repo = local_repo_clone(repo, dir)

local_repo_log(dir)

# Make a local change and push
writeLines("Hello World", file.path(local_repo, "hello.txt"))

local_repo_add(local_repo, "hello.txt")

local_repo_commit(local_repo, "Added hello world")

local_repo_push(local_repo)

repo_commits(repo)

# Pulling remote changes
```

```

repo_modify_file(repo, "hello.txt", pattern = ".*", content = "!!!", method = "after")

local_repo_pull(local_repo)

local_repo_log(dir)

repo_delete("ghclass-test/local_repo_test", prompt=FALSE)

## End(Not run)

```

local_repo_anonymize *Anonymize a local repo or grading project*

Description

[Experimental]

Makes a best-effort pass over a local git repository, a directory of repositories, or a grading project created by [org_grade_assignment\(\)](#) and removes student personally identifying information (PII). Two things happen:

- **File contents** - every text file is scanned and any roster-derived value (names, NetIDs, emails, GitHub usernames, ...) is replaced with a stable per-student token (e.g. student_07).
- **Git history** - each repository's .git directory is either deleted outright (the default) or kept untouched.

Usage

```

local_repo_anonymize(
  path,
  roster,
  cols = tidyselect::any_of(c("name", "first", "last", "email", "netid", "github")),
  output = NULL,
  types = c(".md", ".qmd", ".[Rr]md", ".txt", ".csv", ".tsv", ".html?", ".[Rr]", ".py",
    ".ipynb", ".json", ".ya?ml", ".tex", ".bib", ".toml", ".[Rr]proj"),
  git_history = c("delete", "keep"),
  prompt = interactive()
)

```

Arguments

path	Character. Path to a git repository, a directory of repositories, or a grading project folder (one containing a repos/ subdirectory of cloned repositories, e.g. as produced by org_grade_assignment()).
roster	Either a path to a roster CSV file or a data frame. Each row is treated as one student.

cols	<tidy-select> Roster columns whose values should be stripped from the contents. Defaults to the name, first, last, email, netid, and github columns when present.
output	Character. Directory to write the anonymized copy to. Defaults to <path>_anon. Set output = path to modify the folder in place (gated by a confirmation prompt when prompt = TRUE).
types	Character. Regular expressions matched against the end of each file name to decide which files are treated as text and scanned. Defaults to common text formats found in student work (markdown, R / Python source, notebooks, html, csv, yaml, ...).
git_history	Character. How each repository's git history is handled. Either "delete" (default, remove the .git directory entirely) or "keep" (leave it untouched, retaining the un-anonymized history).
prompt	Logical. Prompt for confirmation before modifying a folder in place. Defaults to <code>interactive()</code> .

Details

Git history cannot be reliably anonymized while also being preserved. Even after rewriting commit author and committer identities, the *original file contents* of older commits remain recoverable (`git show <old-sha>:README.md`), and real names routinely survive in commit messages and branch names. Because anyone you hand the repository to receives its full history, there are only two honest options, selected with `git_history`:

- "delete" (default) - remove each repository's `.git` directory entirely. The result is the scrubbed working tree with no recoverable history. This is the safe choice for sharing student work.
- "keep" - leave `.git` untouched. The working-tree files are still scrubbed, but the original commit history (with real names and emails) is retained, so the result is **not** safe to distribute.

The function works on a *copy* by default (`<path>_anon`), leaving the original intact, so deleting history is non-destructive to your working copy.

Only files whose names match `types` are scanned, so binary and other non-text files are skipped by design. Matching is best effort: word boundaries are used for names / NetIDs to avoid clobbering unrelated text, but short or common names may still over-match, and HTML-entity-encoded names are not detected. Anonymization is roster-driven, so anyone absent from the roster (e.g. a student who later dropped) will not be matched.

Value

Invisibly, a list with two tibbles: `text` (files scrubbed and replacements made per directory) and `git` (the action taken per repo).

Examples

```
## Not run:
local_repo_anonymize(
  "grading/hw1",
```

```

    roster = "rosters/hw1_roster.csv"
  )

  # scrub text but keep history, choosing the columns explicitly
  local_repo_anonymize(
    "grading/hw1",
    roster = roster_df,
    cols = c(name, netid, email),
    git_history = "keep"
  )

  ## End(Not run)

```

local_repo_rename	<i>Rename local directories using a vector of patterns and replacements.</i>
-------------------	--

Description

This function is meant to help with renaming local student repos to include something more useful like Last, First name or a unique identifier for the purposes of ordering repository folders.

Usage

```
local_repo_rename(repo_dir, pattern, replacement)
```

Arguments

repo_dir	Character. A single directory containing one or more repos.
pattern	Character. One or more regexp patterns to match to directory names.
replacement	Character. One or more text strings containing the replacement value for matched patterns.

Value

Returns a character vector of the new repo directory paths, or NA if the rename failed.

Examples

```

## Not run:
# Prefix each cloned repo directory with a roster identifier
local_repo_rename(
  "/path/to/cloned/repos",
  pattern = "hw1-(.*)",
  replacement = "\\1"
)

## End(Not run)

```

org_create_assignment *Create a team or individual assignment*

Description

This is a higher level function that combines the following steps:

- Create repos
- Create teams and invite students if necessary
- Add teams or individuals to the repositories
- Mirror a template repository to assignment repositories

Usage

```
org_create_assignment(
  org,
  repo,
  user,
  team = NULL,
  source_repo = NULL,
  private = TRUE,
  add_badges = FALSE,
  ignore_existing = FALSE
)
```

Arguments

org	Character. Name of the GitHub organization.
repo	Character. Name of the repo(s) for the assignment.
user	Character. GitHub username(s).
team	Character. Team names, if not provided an individual assignment will be created.
source_repo	Character. Address of the repository to use as a template for all created repos.
private	Logical. Should the created repositories be private. Default TRUE.
add_badges	Logical. Should GitHub action badges be added to the README. Default FALSE.
ignore_existing	Logical. If TRUE, any requested repos that already exist are skipped (with a warning) and the remaining repos are still created; useful for resuming a partially failed run. Default FALSE.

Value

An invisible list containing the results of each step.

org_details*Obtain details on an organization's repos and teams*

Description

- `org_exists()` - returns TRUE if the organization(s) exist on GitHub and FALSE otherwise.
- `org_teams()` - returns a (filtered) vector of organization teams.
- `org_team_details()` - returns a data frame of all organization teams containing identification and permission details.
- `org_repos()` - returns a (filtered) vector of organization repositories.
- `org_repo_search()` - search for repositories within an organization (preferred for large organizations).
- `org_repo_stats()` - returns a tibble of repositories belonging to a GitHub organization along with some basic statistics about those repositories.
- `org_repo_forking()` - returns a tibble of private repositories in an organization and whether forking is enabled for each.
- `org_user_repos()` - constructs a character vector of per-user repo addresses (`org/prefix<user>suffix`) aligned with `user`, for safely pairing with `repo_add_user()` and similar.

Usage

```
org_exists(org)
```

```
org_repo_forking(org)
```

```
org_repo_search(org, name, extra = "", full_repo = TRUE)
```

```
org_repo_stats(  
  org,  
  branch = NULL,  
  filter = "",  
  filter_type = "in:name",  
  inc_commits = TRUE,  
  inc_issues = TRUE,  
  inc_prs = TRUE  
)
```

```
org_repos(  
  org,  
  filter = NULL,  
  exclude = FALSE,  
  full_repo = TRUE,  
  sort = c("full_name", "created", "updated", "pushed"),  
  direction = c("asc", "desc"),  
  type = c("all", "public", "private", "forks", "sources", "member", "internal")  
)
```

```

)

org_team_details(org)

org_teams(org, filter = NULL, exclude = FALSE, team_type = c("name", "slug"))

org_user_repos(org, user, prefix = "", suffix = "")

```

Arguments

org	Character. Name of the GitHub organization(s).
name	Character. Full or partial repo name to search for within the org
extra	Character. Any additional search qualifiers, see Searching for repositories for details.
full_repo	Logical. Should the full repository address be returned (e.g. owner/repo instead of just repo). Default TRUE.
branch	Character. The branch to use for counting commits, if NULL then each repo's default branch is used.
filter	Character. Regular expression pattern for matching (or excluding) results
filter_type	Character. One or more GitHub search in qualifiers. See documentation for more details.
inc_commits	Logical. Include commit statistics (branch, commits, last_update). Default TRUE.
inc_issues	Logical. Include issue statistics (open_issues, closed_issues). Default TRUE.
inc_prs	Logical. Include pull request statistics (open_prs, merged_prs, closed_prs). Default TRUE.
exclude	Logical. Should entries matching the regular expression be excluded or included. Default FALSE.
sort	Character. Sorting criteria to use, can be one of "created", "updated", "pushed", or "full_name".
direction	Character. Sorting order to use.
type	Character. Specifies the type of repositories you want, can be one of "all", "public", "private", "forks", "sources", "member", or "internal".
team_type	Character. Either "slug" if the team names are slugs or "name" if full team names are provided.
user	Character. One or more GitHub usernames.
prefix	Character. String to prepend to each user name.
suffix	Character. String to append to each user name.

Value

org_exists() returns a logical vector.

org_teams(), org_repos, org_repo_search(), and org_user_repos() return a character vector.

org_team_details(), org_repo_stats(), and org_repo_forking() return tibbles.

Note

The order of `org_repos()` is not guaranteed to match the order of `org_members()`, so do not rely on positional pairing of the two when calling functions like `repo_add_user()`. To pair each user with their own repo deterministically, build the repo names from the user vector with `org_user_repos()`. `org_repo_search()` and `org_repo_stats()` use GitHub's search API, which returns at most 1000 results per query. For organizations with more than 1000 matching repositories the results are truncated and a warning is issued; use `filter` (or `name / extra`) to narrow the search.

Examples

```
## Not run:
# Org repos and teams
org_repos("ghclass-test")

org_repos("ghclass-test", filter = "hw1-")

org_teams("ghclass-test")

org_team_details("ghclass-test")

## End(Not run)
```

`org_grade_assignment` *Setup grading for an assignment*

Description

This is a higher level function that automates the common steps for setting up grading of an assignment:

- Find repos matching a filter pattern in an organization
- Clone all matched repos locally
- Download GitHub Actions artifacts (e.g. rendered html, md files)
- Report any missing artifacts
- Create comment template files for each repo

Usage

```
org_grade_assignment(
  path,
  org,
  repo_filter,
  artifacts = character(),
  comment_template = "",
  key_repo = NULL
)
```

Arguments

path	Character. Root directory for the grading folder (created if it doesn't exist).
org	Character. Name of the GitHub organization.
repo_filter	Character. Regex pattern passed to <code>org_repos()</code> 's filter argument to select repos (e.g. "hw01_").
artifacts	Named character vector. Names are subfolder names, values are regex patterns passed as filter to <code>action_artifact_download()</code> (matched against artifact names). E.g. <code>c("html" = "html-output", "report" = "pdf-report")</code> .
comment_template	Character. Template text used to populate each comment markdown file. Defaults to "".
key_repo	Character. Optional repository address in owner/name format to clone into the root of the grading folder as an answer key.

Value

An invisible list containing:

- repos — character vector of matched repo addresses
- cloned — result from `local_repo_clone()`
- artifacts — named list of results from each `action_artifact_download()` call
- comments — character vector of comment file paths created
- key — result from cloning the key repo (if provided)

org_members

Tools for managing organization membership

Description

- `org_invite()` - invites user(s) to a GitHub organization.
- `org_remove()` - remove user(s) from an organization (and all teams within that organization).
- `org_members()` - returns a (filtered) vector of organization members.
- `org_pending()` - returns a (filtered) vector of pending organization members.
- `org_admins()` - returns a vector of repository administrators. In the case of a non-organization owner (e.g. a user account) returns the owner's login.

Usage

```

org_admins(org)

org_invite(org, user)

org_members(org, filter = NULL, exclude = FALSE, include_admins = TRUE)

org_pending(org, filter = NULL, exclude = FALSE)

org_remove(org, user, prompt = TRUE)

```

Arguments

org	Character. Name of the GitHub organization(s).
user	Character. GitHub username(s).
filter	Character. Regular expression pattern for matching (or excluding) results
exclude	Logical. Should entries matching the regular expression be excluded or included. Default FALSE.
include_admins	Logical. Should admin users be included in the results. Default TRUE.
prompt	Logical. Prompt before removing member from organization. Default TRUE.

Value

org_members(), org_pending(), and org_admins all return a character vector of GitHub account names.

org_invite() and org_remove() invisibly return a list containing the results of the relevant GitHub API calls.

Examples

```

## Not run:
# Org Details
org_admins("ghclass-test")

org_admins("rundel") # User, not an organization

# Org Membership - Invite, Status, and Remove
students = c("ghclass-anya", "ghclass-bruno", "ghclass-celine",
             "ghclass-diego", "ghclass-elijah", "ghclass-francis")

org_invite("ghclass-test", students)

org_members("ghclass-test")

org_pending("ghclass-test")

org_remove("ghclass-test", students, prompt = FALSE)

```

```
org_pending("ghclass-test")

## End(Not run)
```

org_perm

Organization permissions

Description

- `org_sitrep()` - Provides a situation report on a GitHub organization.
- `org_set_repo_permission()` - Change the default permission level for org repositories.
- `org_workflow_permissions()` - Obtain the current default workflow permission value for the organization.
- `org_set_workflow_permissions()` - Change the current default workflow permission value for the organization.
- `org_allows_forking()` - returns TRUE if members can fork private repositories in the organization.

Usage

```
org_allows_forking(org)

org_sitrep(org)

org_set_repo_permission(
  org,
  repo_permission = c("none", "read", "write", "admin")
)

org_workflow_permissions(org)

org_set_workflow_permissions(org, workflow_permission = c("read", "write"))
```

Arguments

org Character. Name of the GitHub organization(s).

repo_permission Default permission level members have for organization repositories:

- read - can pull, but not push to or administer this repository.
- write - can pull and push, but not administer this repository.
- admin - can pull, push, and administer this repository.
- none - no permissions granted by default.

workflow_permission The default workflow permissions granted to the GITHUB_TOKEN when running workflows in the organization. Accepted values: "read" or "write".

Value

`org_sitep()` invisibly returns the `org` argument.
`org_set_repo_permission()` invisibly return a the result of the relevant GitHub API call.
`org_workflow_permissions()` returns a character vector with value of either "read" or "write".
`org_set_workflow_permissions()` invisibly return a the result of the relevant GitHub API call.
`org_allows_forking()` returns a logical scalar.

Examples

```
## Not run:
org_sitep("ghclass-test")

org_set_repo_permission("ghclass-test", "read")

org_workflow_permissions("ghclass-test")

org_set_workflow_permissions("ghclass-test", "write")

org_sitep("ghclass-test")

# Cleanup
org_set_repo_permission("ghclass-test", "none")
org_set_workflow_permissions("ghclass-test", "read")

## End(Not run)
```

pages

Retrieve information about GitHub Pages sites and builds.

Description

- `pages_enabled()` - returns TRUE if a Pages site exists for the repo.
- `pages_status()` - returns more detailed information about a repo's Pages site.
- `pages_create()` - creates a Pages site for the provided repos.
- `pages_delete()` - deletes the Pages site for the provided repos.

Usage

```
pages_enabled(repo)

pages_status(repo)

pages_create(
  repo,
  build_type = c("legacy", "workflow"),
```

```
    branch = "main",
    path = "/"
  )

  pages_delete(repo, prompt = TRUE)
```

Arguments

repo	Character. Address of repositories in owner/name format.
build_type	Character. Either "workflow" or "legacy" - the former uses GitHub actions to build and publish the site (requires a workflow file to achieve this).
branch	Character. Repository branch to publish.
path	Character. Repository path to publish.
prompt	Logical. Should the user be prompted before deleting Pages sites. Default TRUE.

Value

pages_enabled() returns a named logical vector - TRUE if a Pages site exists, FALSE otherwise.

pages_status() returns a tibble containing details on Pages sites.

pages_create() & pages_delete() return an invisible list containing the API responses.

Examples

```
## Not run:
pages_enabled("rundel/ghclass")

pages_status("rundel/ghclass")

## End(Not run)
```

pr	<i>GitHub Pull Request related tools</i>
----	--

Description

- pr_create() - create a pull request GitHub from the base branch to the head branch.

Usage

```
pr_create(repo, title, head, base, body = "", draft = FALSE)
```

Arguments

repo	Character. Address of one or more repositories in "owner/name" format.
title	Character. Title of the pull request.
head	Character. The name of the branch where your changes are implemented. For cross-repository pull requests in the same network, namespace head with a user like this: username:branch.
base	Character. The name of the branch you want the changes pulled into. This should be an existing branch on the current repository. You cannot submit a pull request to one repository that requests a merge to a base of another repository.
body	Character. The text contents of the pull request.
draft	Logical. Should the pull request be created as a draft pull request (these cannot be merged until allowed by the author). Default FALSE.

Value

pr_create() invisibly return a list containing the results of the relevant GitHub API calls.

See Also

[repo_issues](#)

Examples

```
## Not run:
repo_create("ghclass-test", "test_pr", auto_init=TRUE)

branch_create("ghclass-test/test_pr", branch = "main", new_branch = "test")

repo_modify_file("ghclass-test/test_pr", "README.md", pattern = "test_pr",
                 content = "Hello", method = "after", branch = "test")

pr_create("ghclass-test/test_pr", title = "merge", head = "test", base = "main")

repo_delete("ghclass-test/test_pr", prompt = FALSE)

## End(Not run)
```

repo_core

GitHub Repository tools - core functions

Description

- repo_create() - create a GitHub repository.
- repo_delete() - delete a GitHub repository.
- repo_rename() - rename a repository, note that renamed repositories retain their unique identifier and can still be accessed via their old names due to GitHub re-directing.

- `repo_exists()` - returns TRUE if the GitHub repository exists. It will also print a message if a repository has been renamed, unless `quiet = TRUE`.
- `repo_mirror()` - mirror the content of a repository to another repository, the target repo must already exist.
- `repo_mirror_template()` - mirror the content of a source template repository to a new repository, the target repo must *not* already exist.
- `repo_is_template()` - returns TRUE if a repository is a template repo.
- `repo_set_template()` - change the template status of a repository.
- `repo_allows_forking()` - returns TRUE if forking is enabled for a repository.
- `repo_set_forking()` - enable or disable forking for a repository.

Usage

```
repo_allows_forking(repo)

repo_create(
  org,
  name,
  prefix = "",
  suffix = "",
  private = TRUE,
  auto_init = FALSE,
  gitignore_template = "R"
)

repo_delete(repo, prompt = TRUE)

repo_exists(repo, strict = FALSE, quiet = FALSE)

repo_is_template(repo)

repo_mirror(
  source_repo,
  target_repo,
  overwrite = FALSE,
  verbose = FALSE,
  warn = TRUE
)

repo_mirror_template(source_repo, target_repo, private = TRUE)

repo_rename(repo, new_repo)

repo_set_forking(repo, status = TRUE)

repo_set_template(repo, status = TRUE)
```

Arguments

repo	Character. Address of repository in owner/repo format.
org	Character. GitHub organization that will own the repository
name	Character. Repository name
prefix	Character. Common repository name prefix
suffix	Character. Common repository name suffix
private	Logical. Should the new repository be private or public. Default TRUE.
auto_init	Logical. Should the repository be initialized with a README.md. Default FALSE.
gitignore_template	Character. .gitignore language template to use.
prompt	Logical. Should the user be prompted before deleting repositories. Default true.
strict	Logical. Should the old name of a renamed repositories be allowed. Default FALSE.
quiet	Logical. Should details on renamed repositories be printed. Default FALSE.
source_repo	Character. Address of template repository in owner/name format.
target_repo	Character. One or more repository addresses in owner/name format. Note when using template repos these new repositories must <i>not</i> exist.
overwrite	Logical. Should the target repositories be overwritten. Default FALSE.
verbose	Logical. Display verbose output. Default FALSE.
warn	Logical. Warn the user about the function being deprecated. Default TRUE.
new_repo	Character. New name of repository without the owner.
status	Logical. Should the repository be set as a template repository. Default TRUE.

Value

repo_create() returns a character vector of created repos (in owner/repo format)
 repo_exists(), repo_is_template(), and repo_allows_forking() all return a logical vector.
 repo_set_forking() invisibly returns a list containing the results of the relevant GitHub API calls.
 All other functions invisibly return a list containing the results of the relevant GitHub API calls.

Examples

```
## Not run:
repo_create("ghclass-test", "repo_test")

repo_exists("ghclass-test/repo_test")

repo_rename("ghclass-test/repo_test", "repo_test_new")

# The new repo exists
repo_exists("ghclass-test/repo_test_new")
```

```

# The old repo forwards to the new repo
repo_exists("ghclass-test/repo_test")

# Check for the redirect by setting `strict = TRUE`
repo_exists("ghclass-test/repo_test", strict = TRUE)

# The preferred way of copying a repo is by making the source a template
repo_is_template("ghclass-test/repo_test_new")

repo_set_template("ghclass-test/repo_test_new")

repo_is_template("ghclass-test/repo_test_new")

# Given a template repo we can then directly copy the repo on GitHub
repo_mirror_template("ghclass-test/repo_test_new", "ghclass-test/repo_test_copy")

repo_exists("ghclass-test/repo_test_copy")

# Cleanup
repo_delete(
  c("ghclass-test/repo_test_new",
    "ghclass-test/repo_test_copy"),
  prompt = FALSE
)

## End(Not run)

```

repo_details

GitHub Repository tools - repository details

Description

- `repo_clone_url()` - Returns the url, for cloning, a GitHub repo (either ssh or https)
- `repo_branches()` - Returns a (filtered) vector of branch names.
- `repo_commits()` - Returns a tibble of commits to a GitHub repository.
- `repo_issues()` - Returns a tibble of issues for a GitHub repository.
- `repo_n_commits()` - Returns a tibble of the number of commits in a GitHub repository (and branch).
- `repo_prs()` - Returns a tibble of pull requests for a GitHub repository.
- `repo_pushes()` - Returns a tibble of push activity to a GitHub repository.

Usage

```
repo_branches(repo)

repo_clone_url(repo, type = c("https", "ssh"))

repo_commits(
  repo,
  branch = NULL,
  sha = branch,
  path = NULL,
  author = NULL,
  since = NULL,
  until = NULL,
  quiet = FALSE
)

repo_issues(
  repo,
  state = c("open", "closed", "all"),
  assignee = NULL,
  creator = NULL,
  mentioned = NULL,
  labels = NULL,
  sort = c("created", "updated", "comments"),
  direction = c("desc", "asc"),
  since = NULL
)

repo_n_commits(repo, quiet = FALSE)

repo_prs(repo, state = c("open", "closed", "all"), quiet = FALSE)

repo_pushes(
  repo,
  branch = NULL,
  author = NULL,
  time_period = c("all time", "day", "week", "month", "quarter", "year"),
  quiet = FALSE
)
```

Arguments

repo	Character. Address of repository in owner/name format.
type	Character. Clone url type, either "https" or "ssh".
branch	Character. Branch to list commits from.
sha	Character. SHA to start listing commits from.
path	Character. Only commits containing this file path will be returned.

author	Character. GitHub login or email address by which to filter commit author.
since	Character. Only issues updated at or after this time are returned.
until	Character. Only commits before this date will be returned, expects YYYY-MM-DDTHH:MM:SSZ format. Filtered against the <code>committer_date</code> column, see Details.
quiet	Logical. Should an error message be printed if a repo does not exist. Default FALSE.
state	Character. Pull request state.
assignee	Character. Return issues assigned to a particular username. Pass in "none" for issues with no assigned user, and "*" for issues assigned to any user.
creator	Character. Return issues created the by the given username.
mentioned	Character. Return issues that mentioned the given username.
labels	Character. Return issues labeled with one or more of of the given label names.
sort	Character. What to sort results by. Can be either "created", "updated", or "comments".
direction	Character. The direction of the sort. Can be either "asc" or "desc".
time_period	Character. The time period to filter by. Options are "all time", "day", "week", "month", "quarter", "year".

Details

For `repo_commits()`, GitHub returns two timestamps for each commit: the author date (`date`) and the committer date (`committer_date`). These differ for rebased, cherry-picked, or amended commits. The `since` and `until` arguments are filtered by GitHub against the committer date.

The `login` column is NA when the commit's author email is not linked to a GitHub account. The git author's name and email are still populated in those cases.

Value

`repo_clone_url()` and `repo_branches()` both return a character vector.

`repo_commits()`, `repo_issues()`, `repo_n_commits()`, and `repo_prs()` all return a tibble.

Examples

```
## Not run:
repo_clone_url("rundel/ghclass")

repo_branches("rundel/ghclass")

repo_commits("rundel/ghclass")

repo_issues("rundel/ghclass")

repo_n_commits("rundel/ghclass", branch = "master")

repo_prs("rundel/ghclass")

## End(Not run)
```

Description

- `repo_add_file()` - Add / update files in a GitHub repository. Note that due to delays in caching, files that have been added very recently might not yet be displayed as existing and might accidentally be overwritten.
- `repo_delete_file()` - Delete a file from a GitHub repository
- `repo_modify_file()` - Modify an existing file within a GitHub repository.
- `repo_ls()` - Low level function for listing the files in a GitHub Repository
- `repo_tree()` - Print the file tree of a GitHub repository, similar to `fs::dir_tree()`
- `repo_put_file()` - Low level function for adding a file to a GitHub repository
- `repo_get_file()` - Low level function for retrieving the content of a file from a GitHub Repository
- `repo_get_readme()` - Low level function for retrieving the content of the README.md of a GitHub Repository

Usage

```
repo_add_file(  
  repo,  
  file,  
  message = NULL,  
  repo_folder = NULL,  
  branch = NULL,  
  preserve_path = FALSE,  
  overwrite = FALSE  
)  
  
repo_delete_file(repo, path, message = NULL, branch = NULL)  
  
repo_get_file(repo, path, branch = NULL, quiet = FALSE, include_details = TRUE)  
  
repo_get_readme(repo, branch = NULL, include_details = TRUE)  
  
repo_ls(repo, path = ".", branch = NULL, full_path = FALSE)  
  
repo_modify_file(  
  repo,  
  path,  
  pattern,  
  content,  
  method = c("replace", "before", "after"),  
  all = FALSE,
```

```

    message = "Modified content",
    branch = NULL
)

repo_put_file(
    repo,
    path,
    content,
    message = NULL,
    branch = NULL,
    verbose = TRUE
)

repo_tree(repo, path = ".", branch = NULL, recurse = TRUE)

```

Arguments

repo	Character. Address of repository in owner/name format.
file	Character. Local file path(s) of file or files to be added.
message	Character. Commit message.
repo_folder	Character. Name of folder on repository to save the file(s) to. If the folder does not exist on the repository, it will be created.
branch	Character. Name of branch to use.
preserve_path	Logical. Should the local relative path be preserved. Default FALSE.
overwrite	Logical. Should existing file or files with same name be overwritten. Default FALSE.
path	Character. File's path within the repository.
quiet	Logical. Should status messages be printed. Default FALSE.
include_details	Logical. Should file details be attached as attributes. Default TRUE.
full_path	Logical. Should the function return the full path of the files and directories. Default FALSE.
pattern	Character. Regex pattern.
content	Character or raw. Content of the file.
method	Character. Should the content replace the matched pattern or be inserted before or after the match.
all	Character. Should all instances of the pattern be modified (TRUE) or just the first (FALSE).
verbose	Logical. Should success / failure messages be printed. Default TRUE.
recurse	Logical. Should the listing recurse into subdirectories. Default TRUE.

Value

repo_delete_file(), repo_modify_file(), and repo_put_file() all invisibly return a list containing the results of the relevant GitHub API calls.

repo_ls() returns a character vector of repo files in the given path.

repo_tree() invisibly returns a character vector of the displayed paths.

repo_get_file() and repo_get_readme() return a character vector with API results attached as attributes if include_details = TRUE

Examples

```
## Not run:
repo = repo_create("ghclass-test", "repo_file_test", auto_init=TRUE)

repo_ls(repo, path = ".")

repo_tree(repo)

repo_get_readme(repo, include_details = FALSE)

repo_get_file(repo, ".gitignore", include_details = FALSE)

repo_modify_file(
  repo, path = "README.md", pattern = "repo_file_test",
  content = "\n\nHello world!\n", method = "after"
)

repo_get_readme(repo, include_details = FALSE)

repo_add_file(repo, file = system.file("DESCRIPTION", package="ghclass"))

repo_get_file(repo, "DESCRIPTION", include_details = FALSE)

repo_delete_file(repo, "DESCRIPTION")

repo_delete(repo, prompt=FALSE)

## End(Not run)
```

repo_notification

GitHub Repository tools - notification functions

Description

- repo_ignore() - Ignore a GitHub repository.
- repo_unwatch() - Unwatch / unsubscribe from a GitHub repository.
- repo_watch() - Watch / subscribe to a GitHub repository.
- repo_watching() - Returns a vector of your watched repositories. This should match the list at github.com/watching.

Usage

```

repo_unwatch(repo)

repo_watch(repo)

repo_ignore(repo)

repo_watching(filter = NULL, exclude = FALSE)

```

Arguments

repo	repository address in owner/repo format
filter	character, regex pattern for matching (or excluding) repositories.
exclude	logical, should entries matching the regex be excluded or included.

Value

repo_ignore(), repo_unwatch(), and repo_watch() all invisibly return a list containing the results of the relevant GitHub API call.

repo_watching() returns a character vector of watched repos.

Examples

```

## Not run:
repo_ignore("Sta323-Sp19/hw1")

repo_unwatch("rundel/ghclass")

repo_watch("rundel/ghclass")

## End(Not run)

```

repo_style	<i>Style repository with styler</i>
------------	-------------------------------------

Description

- repo_style implements "non-invasive pretty-printing of R source code" of .R or .Rmd files within a repository using the styler package and adhering to tidyverse formatting guidelines.

Usage

```
repo_style(  
  repo,  
  files = c("*.R", "*.Rmd"),  
  branch = "styler",  
  base,  
  create_pull_request = TRUE,  
  draft = TRUE,  
  tag_collaborators = TRUE,  
  prompt = TRUE  
)
```

Arguments

repo	Character. Address of repository in "owner/name" format.
files	Character or vector of characters. Names of .R and/or .Rmd files that styler should be applied to.
branch	Character. Name of new branch to be created or overwritten. Default is "styler".
base	Character. Name of branch that contains the .R and/or .Rmd files to be styled
create_pull_request	Logical. If TRUE, a pull request is created from branch to base. Default TRUE.
draft	Logical. Should the pull request be created as a draft pull request? (Draft PRs cannot be merged until allowed by the author) Default TRUE.
tag_collaborators	Logical. If TRUE, a message with the repository collaborators is displayed. Default TRUE.
prompt	Logical. Prompt the user before overwriting an existing branch. Default TRUE.

Value

The functions returns NULL invisibly.

Examples

```
## Not run:  
repo_style("ghclass-test/hw1-team01", base = "main")  
  
## End(Not run)
```

Description

- `repo_add_user()` - Add a user to a repository
- `repo_remove_user()` - Remove a user from a repository
- `repo_add_team()` - Add a team to a repository
- `repo_remove_team()` - Remove a team from a repository
- `repo_user_permission()` - Change a collaborator's permissions for a repository
- `repo_team_permission()` - Change a team's permissions for a repository
- `repo_collaborators()` - Returns a data frame of repos, their collaborators, and their permissions.
- `repo_contributors()` - Returns a data frame containing details on repository contributor(s).

Usage

```
repo_add_team(  
  repo,  
  team,  
  permission = c("push", "pull", "admin", "maintain", "triage"),  
  team_type = c("name", "slug")  
)  
  
repo_team_permission(  
  repo,  
  team,  
  permission = c("push", "pull", "admin", "maintain", "triage"),  
  team_type = c("name", "slug")  
)  
  
repo_add_user(  
  repo,  
  user,  
  permission = c("push", "pull", "admin", "maintain", "triage")  
)  
  
repo_user_permission(  
  repo,  
  user,  
  permission = c("push", "pull", "admin", "maintain", "triage")  
)  
  
repo_collaborators(repo, include_admins = TRUE)
```

```

repo_contributors(repo)

repo_remove_team(repo, team, team_type = c("name", "slug"))

repo_remove_user(repo, user)

```

Arguments

repo	Character. Address of repository in owner/repo format.
team	Character. Slug or name of team to add.
permission	Character. Permission to be granted to a user or team for repo, defaults to "push".
team_type	Character. Either "slug" if the team names are slugs or "name" if full team names are provided.
user	Character. One or more GitHub usernames.
include_admins	Logical. If FALSE, user names of users with Admin rights are not included, defaults to TRUE.

Details

Permissions can be set to any of the following:

- "pull" - can pull, but not push to or administer this repository.
- "push" - can pull and push, but not administer this repository.
- "admin" - can pull, push and administer this repository.
- "maintain" - Recommended for project managers who need to manage the repository without access to sensitive or destructive actions.
- "triage" - Recommended for contributors who need to proactively manage issues and pull requests without write access.

repo_contributors() is built on GitHub's /stats/contributors endpoint, which groups commits by GitHub user. Commits whose author email is not linked to a GitHub account are not included in the result.

Value

repo_collaborators() and repo_contributors() return a tibble.
All other functions invisibly return a list containing the results of the relevant GitHub API calls.

Examples

```

## Not run:
repo = repo_create("ghclass-test", "hw1")

team_create("ghclass-test", "team_awesome")

repo_add_user(repo, "rundel")

repo_add_team(repo, "team_awesome")

```

```

repo_remove_team(repo, "team_awesome")

repo_collaborators(repo)

repo_contributors(repo)
repo_contributors("rundel/ghclass")

# Cleanup
repo_delete(repo, prompt=FALSE)

## End(Not run)

```

team	<i>Create, delete, and rename teams within an organization</i>
------	--

Description

- `team_create()` - create teams in a GitHub organization
- `team_delete()` - delete a team from a GitHub organization.
- `team_rename()` - rename an existing team

Usage

```

team_create(
  org,
  team,
  prefix = "",
  suffix = "",
  privacy = c("secret", "closed")
)

team_delete(org, team, team_type = c("name", "slug"), prompt = TRUE)

team_rename(org, team, new_team, team_type = c("name", "slug"))

```

Arguments

org	Character. Name of the GitHub organization.
team	Character. Name of teams.
prefix	Character. Shared prefix.
suffix	Character. Shared suffix.
privacy	Character. Level of privacy for team, "closed" (visible to all members of the organization) or "secret" (only visible to organization owners and members of a team), default is "secret"

team_type	Character. Either "slug" if the team names are slugs or "name" if full team names are provided.
prompt	Logical. Should the user be prompted before deleting team. Default true.
new_team	character, new team name.

Value

All functions invisibly return a list containing the results of the relevant GitHub API calls.

Examples

```
## Not run:
team_create("ghclass-test", c("hw1-team01", "hw1-team02"))

org_teams("ghclass-test", "hw1-")

team_rename("ghclass-test", "hw1-team02", "hw1-team03")

org_teams("ghclass-test", "hw1-")

team_delete("ghclass-test", "hw1-team01", prompt = FALSE)

org_teams("ghclass-test", "hw1-")

# Cleanup
team_delete("ghclass-test", org_teams("ghclass-test", "hw1-"), prompt = FALSE)

## End(Not run)
```

team_members	<i>Tools for inviting, removing, and managing members of an organization team</i>
--------------	---

Description

- team_invite() - add members to team(s).
- team_remove() - remove members from team(s).
- team_members() - returns a tibble of team members.
- team_pending() - returns a tibble of pending team members.
- team_repos() - returns a tibble of teams and their repos.

Usage

```
team_invite(org, user, team, team_type = c("name", "slug"))

team_members(org, team = org_teams(org), team_type = c("name", "slug"))

team_pending(org, team = org_teams(org), team_type = c("name", "slug"))

team_remove(org, user, team, team_type = c("name", "slug"))

team_repos(org, team = org_teams(org), team_type = c("name", "slug"))
```

Arguments

org	Character. Name of the GitHub organization.
user	Character. One or more GitHub users to invite.
team	Character. Name of teams.
team_type	Character. Either "slug" if the team names are slugs or "name" if full team names are provided.

Value

team_members(), team_pending(), and team_repos() all return a tibble. The team column echoes the supplied team values (names or slugs, per team_type) and a slug column gives the corresponding team slug used for the API calls.

team_invite() and team_remove() invisibly return a list containing the results of the relevant GitHub API calls.

Examples

```
## Not run:
team_create("ghclass-test", c("hw1-team01", "hw1-team02"))

team_invite("ghclass-test", user = "rundel", team = c("hw1-team01", "hw1-team02", "missing_team"))

team_remove("ghclass-test", user = "rundel", team = c("hw1-team01", "missing_team"))

team_members("ghclass-test", org_teams("ghclass-test", "hw1-"))

team_pending("ghclass-test", org_teams("ghclass-test", "hw1-"))

# Add team repo
repo_create("ghclass-test", name = "hw1-team02")
repo_add_team("ghclass-test/hw1-team02", team = "hw1-team02")

team_repos("ghclass-test", org_teams("ghclass-test", "hw1-"))

# Cleanup
repo_delete("ghclass-test/hw1-team02", prompt = FALSE)
```

```
team_delete("ghclass-test", org_teams("ghclass-test", "hw1-"), prompt = FALSE)

## End(Not run)
```

team_roster	<i>Add team assignments to a roster</i>
-------------	---

Description

`team_roster()` adds a team (or repo) name column to a roster data frame by randomly grouping members into teams of a target size. Teams may be formed within groups (e.g. lab sections) via `by`, and names are built from a [glue](#) template that can reference any column of roster together with the generated `team_id`. When `by` is supplied and no name is given, the grouping column(s) are folded into the default name so team names stay unique across groups.

Usage

```
team_roster(
  roster,
  size,
  name = NULL,
  by = NULL,
  col = "team",
  pad = TRUE,
  shuffle = TRUE,
  seed = NULL
)
```

Arguments

<code>roster</code>	Data frame. Course roster with one row per student.
<code>size</code>	Integer. Target (maximum) team size. Within each group members are split into $\text{ceiling}(n / \text{size})$ teams of as-equal-as-possible size. An error is thrown if the roster (or any group defined by <code>by</code>) has fewer than <code>size</code> members.
<code>name</code>	Character. A glue template for the team / repo name, used as-is if supplied. May reference any column of roster as well as <code>team_id</code> (the within-group team number). If NULL (default) the template is <code>"team{team_id}"</code> , or <code>"{group}-team{team_id}"</code> when <code>by</code> is set.
<code>by</code>	Character. Optional column name(s) in roster to form teams within (e.g. a lab section). If NULL (default) teams are formed across the whole roster.
<code>col</code>	Character. Name of the output column to add. Default <code>"team"</code> .
<code>pad</code>	Logical or integer. Zero-padding applied to <code>team_id</code> before it is substituted into name. TRUE (default) pads to a consistent width based on the largest team number, FALSE leaves it as an integer, or supply an integer for a fixed width.
<code>shuffle</code>	Logical. Should members be randomly assigned to teams? Default TRUE.
<code>seed</code>	Integer. Optional random seed for reproducible assignments.

Value

The roster data frame with `team_id` and `col` columns added.

Examples

```
## Not run:
roster = readr::read_csv(system.file("roster.csv", package = "ghclass"))

# Teams of 4 across the whole roster -> "team01", "team02", ...
team_roster(roster, size = 4)

# Teams of 4 within each lab section, group folded into the default name
team_roster(roster, size = 4, by = "section", seed = 20250901)

# Custom name template (used exactly as written)
team_roster(
  roster, size = 4, by = "section",
  name = "hw1_lab{section}_team{team_id}", seed = 20250901
)

## End(Not run)
```

user

GitHub user related tools

Description

- `user_exists()` - returns TRUE if the username(s) (or organization) exist on GitHub and FALSE otherwise. Note that GitHub considers organizations to be a type of user.
- `user_repos()` - returns a (filtered) vector of repositories belonging to the user.
- `user_type()` - returns a vector of the accounts' types.

Usage

```
user_exists(user)

user_repos(
  user,
  type = c("owner", "all", "public", "private", "member"),
  filter = NULL,
  exclude = FALSE,
  full_repo = TRUE
)

user_type(user)
```

Arguments

user	Character. GitHub username(s).
type	Character. Can be one of "all", "owner", "public", "private", "member".
filter	Character. Regular expression pattern for matching (or excluding) repositories.
exclude	Logical. Should entries matching the regular expression in filter be excluded or included? Default FALSE.
full_repo	Logical. Should the full repository address be returned (e.g. owner/repo instead of just repo)? Default TRUE.

Value

user_exists() returns a logical vector.

user_repos() and user_type() return a character vector.

Examples

```
## Not run:
user_exists(c("rundel", "ghclass-test", "hopefullydoesnotexist"))

user_repos("rundel", type = "public", filter = "ghclass")

user_repos("ghclass-test")

org_repos("ghclass-test")

user_type(c("rundel", "ghclass-test"))

## End(Not run)
```

Index

action, [2](#)
action_add_badge (action_badge), [5](#)
action_artifact_delete (action), [2](#)
action_artifact_download (action), [2](#)
action_artifact_download(), [24](#)
action_artifacts (action), [2](#)
action_badge, [5](#)
action_remove_badge (action_badge), [5](#)
action_run_logs (action), [2](#)
action_runs (action), [2](#)
action_runtime (action), [2](#)
action_status (action), [2](#)
action_workflows (action), [2](#)

branch, [6](#)
branch_create (branch), [6](#)
branch_delete (branch), [6](#)
branch_remove (branch), [6](#)

gh::gh(), [8](#)
gh::gh_token(), [10](#)
github_api_limit, [7](#)
github_get_api_limit
 (github_api_limit), [7](#)
github_get_max_rate (github_api_limit),
 [7](#)
github_get_max_wait (github_api_limit),
 [7](#)
github_get_token (github_rate_limit), [9](#)
github_graphql_rate_limit
 (github_rate_limit), [9](#)
github_orgs, [9](#)
github_rate_limit, [9](#)
github_reset_token (github_rate_limit),
 [9](#)
github_set_api_limit
 (github_api_limit), [7](#)
github_set_max_rate (github_api_limit),
 [7](#)
github_set_max_wait (github_api_limit),
 [7](#)
github_set_token (github_rate_limit), [9](#)
github_test_token (github_rate_limit), [9](#)
github_token (github_rate_limit), [9](#)
github_token_scopes
 (github_rate_limit), [9](#)
github_token_sitrep
 (github_rate_limit), [9](#)
github_whoami, [11](#)
github_with_pat, [12](#)
glue, [45](#)

interactive(), [18](#)
issue, [13](#)
issue_close (issue), [13](#)
issue_create (issue), [13](#)
issue_edit (issue), [13](#)

local_pat (github_with_pat), [12](#)
local_repo, [15](#)
local_repo_add (local_repo), [15](#)
local_repo_anonymize, [17](#)
local_repo_branch (local_repo), [15](#)
local_repo_clone (local_repo), [15](#)
local_repo_clone(), [24](#)
local_repo_commit (local_repo), [15](#)
local_repo_log (local_repo), [15](#)
local_repo_pull (local_repo), [15](#)
local_repo_push (local_repo), [15](#)
local_repo_rename, [19](#)

org_admins (org_members), [24](#)
org_allows_forking (org_perm), [26](#)
org_create_assignment, [20](#)
org_details, [21](#)
org_exists (org_details), [21](#)
org_grade_assignment, [23](#)
org_grade_assignment(), [17](#)
org_invite (org_members), [24](#)

org_members, 24
org_pending (org_members), 24
org_perm, 26
org_remove (org_members), 24
org_repo_forking (org_details), 21
org_repo_search (org_details), 21
org_repo_stats (org_details), 21
org_repos (org_details), 21
org_repos(), 24
org_set_repo_permission (org_perm), 26
org_set_workflow_permissions
 (org_perm), 26
org_sitrep (org_perm), 26
org_team_details (org_details), 21
org_teams (org_details), 21
org_user_repos (org_details), 21
org_workflow_permissions (org_perm), 26

pages, 27
pages_create (pages), 27
pages_delete (pages), 27
pages_enabled (pages), 27
pages_status (pages), 27
pr, 28
pr_create (pr), 28

repo_add_file (repo_file), 35
repo_add_team (repo_user), 40
repo_add_user (repo_user), 40
repo_add_user(), 21, 23
repo_allows_forking (repo_core), 29
repo_branches, 7
repo_branches (repo_details), 32
repo_clone (local_repo), 15
repo_clone_url (repo_details), 32
repo_collaborators (repo_user), 40
repo_commits (repo_details), 32
repo_contributors (repo_user), 40
repo_core, 29
repo_create (repo_core), 29
repo_delete (repo_core), 29
repo_delete_file (repo_file), 35
repo_details, 32
repo_exists (repo_core), 29
repo_file, 35
repo_get_file (repo_file), 35
repo_get_readme (repo_file), 35
repo_ignore (repo_notification), 37
repo_is_template (repo_core), 29
repo_issues, 14, 29
repo_issues (repo_details), 32
repo_ls (repo_file), 35
repo_mirror (repo_core), 29
repo_mirror_template (repo_core), 29
repo_modify_file (repo_file), 35
repo_n_commits (repo_details), 32
repo_notification, 37
repo_prs (repo_details), 32
repo_pushes (repo_details), 32
repo_put_file (repo_file), 35
repo_remove_team (repo_user), 40
repo_remove_user (repo_user), 40
repo_rename (repo_core), 29
repo_set_forking (repo_core), 29
repo_set_template (repo_core), 29
repo_style, 38
repo_team_permission (repo_user), 40
repo_tree (repo_file), 35
repo_unwatch (repo_notification), 37
repo_user, 40
repo_user_permission (repo_user), 40
repo_watch (repo_notification), 37
repo_watching (repo_notification), 37

team, 42
team_create (team), 42
team_delete (team), 42
team_invite (team_members), 43
team_members, 43
team_pending (team_members), 43
team_remove (team_members), 43
team_rename (team), 42
team_repos (team_members), 43
team_roster, 45

user, 46
user_exists (user), 46
user_repos (user), 46
user_type (user), 46

with_pat (github_with_pat), 12