Package ‘projmgr’

August 5, 2019

Title Task Tracking and Project Management with GitHub

Version 0.1.0

Description Provides programmatic access to 'GitHub' API with a focus on project management. Key functionality includes setting up issues and milestones from R objects or 'YAML' configurations, querying outstanding or completed tasks, and generating progress updates in tables, charts, and RMarkdown reports. Useful for those using 'GitHub' in personal, professional, or academic settings with an emphasis on streamlining the workflow of data analysis projects.

License MIT + file LICENSE

URL https://github.com/emilyriederer/projmgr

BugReports https://github.com/emilyriederer/projmgr/issues

Depends R (>= 3.1.2)

Imports gh, magrittr

Suggests clipr, curl, dplyr, ggplot2, knitr, purrr, reprex, rmarkdown, testthat, tidyr, yaml, htmltools, httr, covr

Encoding UTF-8

Language en-US

LazyData true

RoxygenNote 6.1.1

NeedsCompilation no

Author Emily Riederer [cre, aut]

Maintainer Emily Riederer <emilyriederer@gmail.com>

Repository CRAN

Date/Publication 2019-08-05 15:40:06 UTC
### R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>browse_docs</td>
<td>3</td>
</tr>
<tr>
<td>browse_issues</td>
<td>3</td>
</tr>
<tr>
<td>browse_milestones</td>
<td>4</td>
</tr>
<tr>
<td>browse_repo</td>
<td>5</td>
</tr>
<tr>
<td>check_credentials</td>
<td>5</td>
</tr>
<tr>
<td>check_internet</td>
<td>6</td>
</tr>
<tr>
<td>check_rate_limit</td>
<td>6</td>
</tr>
<tr>
<td>create_repo_ref</td>
<td>7</td>
</tr>
<tr>
<td>get_issues</td>
<td>8</td>
</tr>
<tr>
<td>get_issue_comments</td>
<td>9</td>
</tr>
<tr>
<td>get_issue_events</td>
<td>10</td>
</tr>
<tr>
<td>get_milestones</td>
<td>10</td>
</tr>
<tr>
<td>get_repo_labels</td>
<td>11</td>
</tr>
<tr>
<td>help</td>
<td>12</td>
</tr>
<tr>
<td>listcol_extract</td>
<td>13</td>
</tr>
<tr>
<td>listcol_filter</td>
<td>14</td>
</tr>
<tr>
<td>listcol_pivot</td>
<td>15</td>
</tr>
<tr>
<td>parse_issues</td>
<td>16</td>
</tr>
<tr>
<td>parse_issue_comments</td>
<td>16</td>
</tr>
<tr>
<td>parse_issue_events</td>
<td>17</td>
</tr>
<tr>
<td>parse_milestones</td>
<td>18</td>
</tr>
<tr>
<td>parse_repo_labels</td>
<td>19</td>
</tr>
<tr>
<td>post_issue</td>
<td>19</td>
</tr>
<tr>
<td>post_milestone</td>
<td>20</td>
</tr>
<tr>
<td>post_plan</td>
<td>21</td>
</tr>
<tr>
<td>post_todo</td>
<td>22</td>
</tr>
<tr>
<td>projmgr</td>
<td>23</td>
</tr>
<tr>
<td>read_plan</td>
<td>23</td>
</tr>
<tr>
<td>read_todo</td>
<td>24</td>
</tr>
<tr>
<td>report_discussion</td>
<td>25</td>
</tr>
<tr>
<td>report_plan</td>
<td>26</td>
</tr>
<tr>
<td>report_progress</td>
<td>27</td>
</tr>
<tr>
<td>report_taskboard</td>
<td>28</td>
</tr>
<tr>
<td>report_todo</td>
<td>29</td>
</tr>
<tr>
<td>taskboard_helpers</td>
<td>30</td>
</tr>
<tr>
<td>template_yaml</td>
<td>32</td>
</tr>
<tr>
<td>viz_gantt</td>
<td>32</td>
</tr>
<tr>
<td>viz_taskboard</td>
<td>34</td>
</tr>
<tr>
<td>viz_waterfall</td>
<td>35</td>
</tr>
</tbody>
</table>

### Index

37
**browse_docs**

View GitHub API documentation

**Description**

Opens browser to relevant parts of GitHub API documentation to learn more about field definitions and formatting. Inspired by similar browse_ functions included in the usethis package.

**Usage**

```r
browse_docs(action = c("get", "post"), object = c("milestone", "issue", "issue event", "issue comment", "repo labels"))
```

**Arguments**

- **action**: Character string denoting action you wish to complete: "get" (list existing) or "post" (creating new)
- **object**: Character string denoting object on wish you want to apply an action. Supports "milestone", "issue", "issue event"

**Value**

Returns URL in non-interactive session or launches browser to docs in interactive session

**Examples**

```r
## Not run:
browse_docs("get", "milestone")
## End(Not run)
```

---

**browse_issues**

Browse issues for given GitHub repo

**Description**

Opens browser to GitHub issues for a given repo. Inspired by similar browse_ functions included in the usethis package.

**Usage**

```r
browse_issues(repo_ref, number = "")
```

**Arguments**

- **repo_ref**: Repository reference as created by create_repo_ref()
- **number**: Optional argument of issue number, if opening page for specific issue is desired
browse_milestones

Value

Returns URL in non-interactive session or launches browser to docs in interactive session

Examples

```r
## Not run:
my_repo <- create_repo_ref("repo_owner", "repo")
browse_issues(my_repo)

## End(Not run)
```

---

**browse_milestones**

Browse milestones for given GitHub repo

Description

Opens browser to GitHub milestones for a given repo. Inspired by similar browse_ functions included in the usethis package.

Usage

```r
browse_milestones(repo_ref, number = "")
```

Arguments

- `repo_ref`: Repository reference as created by `create_repo_ref()`
- `number`: Optional argument of milestone ID, if opening page for specific milestone is desired

Value

Returns URL in non-interactive session or launches browser to docs in interactive session

Examples

```r
## Not run:
my_repo <- create_repo_ref("repo_owner", "repo")
browse_milestones(my_repo)

## End(Not run)
```
**browse_repo**

Browse a given GitHub repo

**Description**

Opens browser to a given GitHub repo. Inspired by similar browse_ functions included in the usethis package.

**Usage**

browse_repo(repo_ref)

**Arguments**

- **repo_ref**: Repository reference as created by create_repo_ref()

**Value**

Returns URL in non-interactive session or launches browser to docs in interactive session

**Examples**

```r
## Not run:
my_repo <- create_repo_ref("repo_owner", "repo")
browse_repo(my_repo)

## End(Not run)
```

---

**check_credentials**

Check for valid credentials and repo permissions

**Description**

Check for valid credentials and repo permissions

**Usage**

check_credentials(ref)

**Arguments**

- **ref**: Any repository reference being used. Repository information is stripped out and only authentication credentials are validated.

**Value**

Prints GitHub username as determined by credentials (if valid) and repo-level permissions (if any), else throws 401 Unauthorized error.
check_rate_limit

Examples

## Not run:
experigit <- create_repo_ref('emilyriederer', 'experigit')
check_authentication(experigit)

## End(Not run)

check_internet  Check internet connection (re-export of curl::has_internet())

Description

Basic wrapper around curl::has_internet()

Usage

check_internet()

Value

Returns TRUE is connected to internet and false otherwise

See Also

Other check: check_rate_limit

Examples

## Not run:
check_internet()

## End(Not run)

check_rate_limit  Find requests remaining and reset time

Description

Source: copied from httr vignette "Best practices for API packages" by Hadley Wickham

Usage

check_rate_limit(ref)
create_repo_ref

Arguments

ref Any repository reference being used. Repository information is stripped out and only authentication credentials are used to determine the rate limit.

Value

Informative message on requests remaining and reset time

See Also

Other check: check_internet

Examples

```r
## Not run:
experigit <- create_repo_ref('emilyriederer', 'experigit')
check_rate_limit(experigit)

## End(Not run)
```

create_repo_ref  Create reference to a GitHub repository

Description

This function constructs a list of needed information to send API calls to a specific GitHub repository. Specifically, it stores information on the repository’s name and owner, the type (whether or not Enterprise GitHub), and potentially credentials to authenticate.

Usage

```r
create_repo_ref(repo_owner, repo_name, is_enterprise = FALSE, hostname = "", identifier = "")
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repo_owner</td>
<td>Repository owner’s username or GitHub Organization name</td>
</tr>
<tr>
<td>repo_name</td>
<td>Repository name</td>
</tr>
<tr>
<td>is_enterprise</td>
<td>Boolean denoting whether or not working with Enterprise GitHub. Defaults to FALSE</td>
</tr>
<tr>
<td>hostname</td>
<td>Host URL stub for Enterprise repositories (e.g. &quot;mycorp.github.com&quot;)</td>
</tr>
<tr>
<td>identifier</td>
<td>Ideally should be left blank and defaults to using GITHUB_PAT or GITHUB_ENT_PAT environment variables as Personal Access Tokens. If identifier, this is assumed to be an alternative name of the environment variable to use for your Personal Access Token</td>
</tr>
</tbody>
</table>
get_issues

Details

Note that this package can be used for GET requests on public repositories without any authentication (resulting in a lower rate limit.) To do this, simply pass any string into identifier that is not an environment variable already defined for your system (e.g. accessible through Sys.getenv("MY_VAR"))

Value

List of repository reference information and credentials

Examples

```r
## Not run:
myrepo <- create_repo_ref('emilyriederer', 'myrepo')

## End(Not run)
```

---

get_issues  

*Get issues from GitHub repository*

Description

A single issue can be obtained by identification number of number is passed through ...s. In this case, all other query parameters will be ignored.

Usage

`get_issues(ref, limit = 1000, ...)`

Arguments

- `ref`: Repository reference (list) created by `create_repo_ref()`
- `limit`: Number of records to return, passed directly to gh documentation. Defaults to 1000 and provides message if number of records returned equals the limit
- `...`: Additional user-defined query parameters. Use `browse_docs()` to learn more.

Value

Content of GET request as list

See Also

Other issues: `get_issue_comments`, `get_issue_events`, `parse_issue_comments`, `parse_issue_events`, `parse_issues`, `post_issue`, `report_discussion`, `report_progress`, `viz_waterfall`
get_issue_comments

Examples

```r
## Not run:
myrepo <- create_repo_ref('emilyriederer', 'myrepo')
issues_res <- get_issues(myrepo)
issues <- parse_issues(issues_res)

## End(Not run)
```

get_issue_comments Get comments for a specific issue from GitHub repository

Description

In addition to information returned by GitHub API, appends field "number" for the issue number to which the returned comments correspond.

Usage

```r
get_issue_comments(ref, number, ...)
```

Arguments

- `ref` Repository reference (list) created by `create_repo_ref()`
- `number` Number of issue
- `...` Additional user-defined query parameters. Use `browse_docs()` to learn more.

Value

Content of GET request as list

See Also

Other issues: `get_issue_events`, `get_issues`, `parse_issue_comments`, `parse_issue_events`, `parse_issues`, `post_issue`, `report_discussion`, `report_progress`, `viz_waterfall`

Other comments: `parse_issue_comments`, `report_discussion`

Examples

```r
## Not run:
myrepo <- create_repo_ref('emilyriederer', 'myrepo')
comments_res <- get_issue_comments(myrepo, number = 1)
comments <- parse_issue_comments(comments_res)

## End(Not run)
```
get_issue_events

Get events for a specific issue from GitHub repository

Description

In addition to information returned by GitHub API, appends field "number" for the issue number to which the returned events correspond.

Usage

get_issue_events(ref, number)

Arguments

ref  Repository reference (list) created by create_repo_ref()
number  Number of issue

Value

Content of GET request as list

See Also

Other issues: get_issue_comments, get_issues, parse_issue_comments, parse_issue_events, parse_issues, post_issue, report_discussion, report_progress, viz_waterfall

Other events: parse_issue_events

Examples

## Not run:
myrepo <- create_repo_ref('emilyriederer', 'myrepo')
events_res <- get_issue_events(myrepo, number = 1)
events <- parse_issue_events(events_res)
## End(Not run)

get_milestones

Get milestones from GitHub repository

Description

A single milestone can be obtained by identification number of number is passed through ...s. In this case, all other query parameters will be ignored.

Usage

get_milestones(ref, ...)

get_repo_labels

Get all labels for a repository

Description

Get all labels for a repository

Usage

get_repo_labels(ref)

Arguments

ref Repository reference (list) created by create_repo_ref()

Value

Content of GET request as list

See Also

Other labels: parse_repo_labels
Examples

```r
## Not run:
labels_res <- get_repo_labels(my_repo)
labels <- parse_repo_labels(labels_res)
## End(Not run)
```

Description

The `help` family of functions lists the optional query parameters available for each of the related `get_*` functions. When no optional arguments are available, a blank character vector is returned.

Usage

- `help_get_issues()`
- `help_get_issue_events()`
- `help_get_issue_comments()`
- `help_get_milestones()`
- `help_get_repo_label()`
- `help_post_issue()`
- `help_post_milestone()`

Details

For more details on these parameters, please use the `browse_docs()` function to navigate to the appropriate part of the GitHub API documentation.

Value

Character string of optional field names

Examples

```r
help_get_issues()
help_get_milestones()
```
**listcol_extract**

*Extract new dataframe column from list-column matching pattern*

**Description**

Creates a new column in your dataframe based on a subset of list-column values following a certain pattern. For example, this is useful if you have labels you always apply to a repository with a set structure, e.g. key-value pairs like "priority:high", "priority:medium", and "priority:low" or other structures like "engagement-team", "teaching-team", etc. This function could create a new variable (e.g. "priority", "team") with the values encoded within the labels.

**Usage**

```r
listcol_extract(data, col_name, regex, new_col_name = NULL, keep_regex = FALSE)
```

**Arguments**

- `data` Dataframe containing a list column (e.g. an issues dataframe)
- `col_name` Character string containing column name of list column (e.g. `labels_name` or `assignees_login`)
- `regex` Character string of regular expression to identify list items of interest (e.g. "^priority:","^(bug|feature)"
- `new_col_name` Optional name of new column. Otherwise `regex` is used, stripped of any leading or trailing punctuation
- `keep_regex` Optional logical denoting whether to keep regex part of matched item in value. Defaults to `FALSE`

**Details**

This function works only if each observation contains at most one instance of a given pattern. When multiple labels match the same pattern, one is returned at random.

**Value**

Dataframe with new column taking values extracted from list column

**Examples**

```r
## Not run:
issues <- get_issues(repo)
issues_df <- parse_issues(issues)
listcol_extract(issues_df, "labels_name", "$-team$"

## End(Not run)
```
listcol_filter  
Filter dataframe by list-column elements

Description

Some outputs of the `get_` and `parse_` functions contain list-columns (e.g. the labels column in the issues dataframe). This is an efficient way to represent the provided information, but may make certain information seem slightly inaccessible. This function allows users to filter list columns by the presence of one or more values or a regular expression.

Usage

```r
listcol_filter(data, col_name, matches, is_regex = FALSE)
```

Arguments

- `data` : Dataframe containing a list column (e.g. an issues dataframe)
- `col_name` : Character string containing column name of list column (e.g. `labels_name` or `assignees_login`)
- `matches` : A character vector containing a regular expression or one or more exact-match values. An observation will be kept in the returned data if any of the
- `is_regex` : Logical to indicate whether character indicates a regular expression or specific values

Value

Dataframe containing only rows in which list-column contains element matching provided criteria

Examples

```r
## Not run:
issues <- get_issues(repo)
issues_df <- parse_issues(issues)

# keep observation containing a label of either "bug" or "feature"
listcol_filter(issues_df, col_name = "labels_name", matches = c("bug", "feature"))

# keep observation containing a label that starts with "region"
listcol_filter(issues_df, col_name = "labels_name", matches = "^region:", is_regex = TRUE)
```

## End(Not run)
listcol_pivot

Pivot list-column elements to indicator variables

Description
Some outputs of the get_ and parse_ functions contain list-columns (e.g. the labels column in the issues dataframe). This is an efficient way to represent the provided information, but may make certain information seem slightly inaccessible. This function allows users to "pivot" these list columns and instead create a separate indicator variable to represent the presence or absence of matches within the list column.

Usage

listcol_pivot(data, col_name, regex = "."., transform_fx = identity, delete_orig = FALSE)

Arguments

data            Dataframe containing a list column (e.g. an issues dataframe)
col_name        Character string containing column name of list column (e.g. labels_name or assignees_login)
regex           Character string of regular expression to identify list items of interest (e.g. "^priority:","^(bug|feature)"
transform_fx    Function to transform label name before converting to column (e.g. sub(":\","_"))
delete_orig     Logical denoting whether or not to delete original list column provided by col_name

Details
For example, if a repository tags issues with "priority:high", "priority:medium", and "priority:low" along with other labels, this function could be used to create separate "high", "medium", and "low" columns to denote different issue severities. This could be done with listcol_pivot(data,"labels_name","^priority:"
sub("^priority:",""))

Value
Dataframe additional logical columns denoting absence / presence of specified list-column elements

Examples

## Not run:
issues <- get_issues(repo)
issues_df <- parse_issues(issues)
listcol_pivot(issues_df,  
col_name = "labels_name",  
regex = "^priority:",  
transform_fx = function(x) paste0("label_",x),  
delete_orig = TRUE)

## End(Not run)
parse_issues

Parse issues overview from get_issues

Description

Parse issues overview from get_issues

Usage

parse_issues(res)

Arguments

res List returned by corresponding get_function

Value
data.frame with one record / issue

See Also

Other issues: get_issue_comments, get_issue_events, get_issues, parse_issue_comments, parse_issue_events, post_issue, report_discussion, report_progress, viz_waterfall

Examples

## Not run:
myrepo <- create_repo_reference('emilyriederer', 'myrepo')
issues_res <- get_issues(myrepo)
issues <- parse_issues(issues_res)

## End(Not run)

parse_issue_comments

Parse issue comments from get_issues_comments

Description

Parse issue comments from get_issues_comments

Usage

parse_issue_comments(res)

Arguments

res List returned by corresponding get_function
parse_issue_events

Value

Dataframe with one record / issue-comment

See Also

Other issues: get_issue_comments, get_issue_events, get_issues, parse_issue_events, parse_issues, post_issue, report_discussion, report_progress, viz_waterfall

Other comments: get_issue_comments, report_discussion

Examples

```r
## Not run:
myrepo <- create_repo_ref('emilyriederer', 'myrepo')
comments_res <- get_issue_comments(myrepo, number = 1)
comments <- parse_issue_comments(comments_res)

## End(Not run)
```

parse_issue_events Parse issue events from get_issues_events

Description

This function convert list output returned by get into a dataframe. Due to the diverse fields for different types of events, many fields in the dataframe may be NA.

Usage

```r
parse_issue_events(res)
```

Arguments

res List returned by corresponding get_ function

Details

Currently, the following event types are unsupported (with regard to processing all of their fields) due to their additional bulk and limited utility with respect to this packages functionality. Please file an issue if you disagree:

- "(removed_from/moved_columns_in/added_to)_project" Since this package has limited value with GitHub projects
- "converted_note_to_issue" Since issue lineage is not a key concern
- "head_ref_(deleted/restored)" Since future support for pull requests would likely be handled separately
- "merged" Same justification as head_ref
- "review_(requested/dismissed/request_removed)" Same justification as head_ref
parse_milestones

Value

Dataframe with one record / issue-event

See Also

Other issues: `get_issue_comments`, `get_issue_events`, `get_issues`, `parse_issue_comments`, `parse_issues`, `post_issue`, `report_discussion`, `report_progress`, `viz_waterfall`
Other events: `get_issue_events`

Examples

```r
## Not run:
myrepo <- create_repo_ref("emilyriederer", "myrepo")
events_res <- get_issue_events(myrepo, number = 1)
events <- parse_issue_events(events_res)
## End(Not run)
```

---

parse_milestones Parse milestones from get_milestones

Description

Parse milestones from get_milestones

Usage

```r
parse_milestones(res)
```

Arguments

`res` List returned by corresponding get_ function

Value

Dataframe with one record / milestone

See Also

Other milestones: `get_milestones`, `post_milestone`

Examples

```r
## Not run:
myrepo <- create_repo_ref("emilyriederer", "myrepo")
milestones_res <- get_milestones(myrepo)
milestones <- parse_milestones(milestones_res)
## End(Not run)
```
**parse_repo_labels**  
*Parse labels from get_repo_labels*

### Description

Parse labels from get_repo_labels

### Usage

```
parse_repo_labels(res)
```

### Arguments

- `res`  
  List returned by corresponding `get_` function

### Value

Dataframe with one record / label

### See Also

Other labels: `get_repo_labels`

### Examples

```r
## Not run:
labels_res <- get_repo_labels(my_repo)
labels <- parse_repo_labels(labels_res)
## End(Not run)
```

---

**post_issue**  
*Post issue to GitHub repository*

### Description

Post issue to GitHub repository

### Usage

```
post_issue(ref, title, ..., distinct = TRUE)
```
Arguments

ref  Repository reference (list) created by create_repo_ref()


title  Issue title (required)

...  Additional user-defined body parameters. Use browse_docs() to learn more.

distinct  Logical value to denote whether issues with the same title as a current open issue

should be allowed

Value

Number (identifier) of posted issue

See Also

Other issues: get_issue_comments, get_issue_events, get_issues, parse_issue_comments, parse_issue_events, parse_issues, report_discussion, report_progress, viz_waterfall

Examples

## Not run:
myrepo <- create_repo_ref('emilyriederer', 'myrepo')
post_issue(myrepo,
    title = 'this is the issue title',
    body = 'this is the issue body',
    labels = c('priority:high', 'bug'))

## End(Not run)

## Not run:
# can be used in conjunction with reprex pkg
# example assumes code for reprex is on clipboard
reprex::reprex(venue = "gh")
post_issue(myrepo,
    title = "something is broken",
    body = paste( clipr::read_clip(), collapse = "\n"))

## End(Not run)
post_plan

Arguments

ref Repository reference (list) created by create_repo_ref()

Arguments

title Milestone title (required)

... Additional user-defined body parameters. Use browse_docs() to learn more.

Value

Number (identifier) of posted milestone

See Also

Other milestones: get_milestones, parse_milestones

Examples

```r
## Not run:
myrepo <- create_repo_ref('emilyriederer', 'myrepo')
post_milestone(myrepo,
  title = 'this is the title of the milestone',
  description = 'this is the long and detailed description',
  due_on = '2018-12-31T12:59:59z')
## End(Not run)
```

Description

Post custom plans (i.e. create milestones and issues) based on yaml read in by read_plan. Please see the "Building Custom Plans" vignette for details.

Usage

```r
post_plan(ref, plan, distinct = TRUE)
```

Arguments

ref Repository reference (list) created by create_repo_ref()

Arguments

plan Plan list as read with read_plan()

distinct Logical value to denote whether issues with the same title as a current open issue should be allowed. Passed to get_issues()

Value

Dataframe with numbers (identifiers) of posted milestones and issues and issue title
See Also

Other plans and todos: `post_todo, read_plan, read_todo, report_plan, report_todo, template_yaml`

Examples

```r
## Not run:
# This example uses example file included in pkg
# You should be able to run example as-is after creating your own repo reference
file_path <- system.file("extdata", "plan.yml", package = "projmgr", mustWork = TRUE)
my_plan <- read_plan(file_path)
post_plan(ref, my_plan)

## End(Not run)
```

---

### post_todo

**Post to-do list (issues) to GitHub repository**

---

**Description**

Post custom to-do lists (i.e. issues) based on yaml read in by `read_todo`. Please see the "Building Custom Plans" vignette for details.

**Usage**

```r
post_todo(ref, todo, distinct = TRUE)
```

**Arguments**

- `ref`: Repository reference (list) created by `create_repo_ref()`
- `todo`: To-do R list structure as read with `read_todo()`
- `distinct`: Logical value to denote whether issues with the same title as a current open issue should be allowed. Passed to `get_issues()`

**Details**

Currently has know bug in that cannot be used to introduce new labels.

**Value**

Number (identifier) of posted issue

**See Also**

Other plans and todos: `post_plan, read_plan, read_todo, report_plan, report_todo, template_yaml`
Examples

```r
# Not run:
# This example uses example file included in pkg
# You should be able to run example as-is after creating your own repo reference
file_path <- system.file("extdata", "todo.yml", package = "projmgr", mustWork = TRUE)
my_todo <- read_todo(file_path)
post_todo(ref, my_todo)

# End(Not run)
```

projmgr

**projmgr package**

Description

A quick and easy wrapper for working with GitHub and other project management tools

Details

See the README on GitHub

read_plan

*Read plan from YAML*

Description

This function reads a carefully constructed YAML file representing a project plan (of milestones and issues). YAML is converted into an R list structure which can then be passed to `post_plan()` to build infrastructure for your repository.

Usage

```r
read_plan(input)
```

Arguments

- `input` Either filepath to YAML file or character string. Assumes filepath if ends in ".yml" and assumes string otherwise.

Details

Please see the "Building Custom Plans" vignette for more details.

Value

List containing plan compatible with `post_plan()` or `post_todo()`
read_todo

See Also

Other plans and todos: post_plan, post_todo, read_todo, report_plan, report_todo, template_yaml

Examples

```r
## Not run:
# This example uses example file included in pkg
# You should be able to run example as-is after creating your own repo reference
file_path <- system.file("extdata", "plan.yml", package = "projmgr", mustWork = TRUE)
my_plan <- read_plan(file_path)
post_plan(ref, my_plan)

## End(Not run)
```

---

**read_todo**  
*Read to-do list from YAML*

Description

This function reads a carefully constructed YAML file representing a to-do list (of issues). YAML is converted into an R list structure which can then be passed to post_todo() to build infrastructure for your repository.

Usage

```r
read_todo(input)
```

Arguments

- `input`  
  Either filepath to YAML file or character string. Assumes filepath if ends in ".yml" and assumes string otherwise.

Details

Please see the "Building Custom Plans" vignette for more details.

Value

List containing plan compatible with post_plan() or post_todo()
### Examples

```r
## Not run:
# This example uses example file included in pkg
# You should be able to run example as-is after creating your own repo reference
file_path <- system.file("extdata", "todo.yml", package = "projmgr", mustWork = TRUE)
my_todo <- read_todo(file_path)
post_todo(ref, my_todo)

## End(Not run)
```

---

**Description**

Interprets dataframe or tibble of issues by breaking apart milestones and listing each issue title as open or closed, and uses HTML to format results in a highly readable and attractive way. Resulting object returned is a character vector of HTML code with the added class of `"knit_asis"` so that when included in an RMarkdown document knitting to HTML, the results will be correctly rendered as HTML.

**Usage**

```r
report_discussion(comments, issue = NA, link_url = TRUE)
```

**Arguments**

- **comments**: Dataframe or tibble of comments for a single issue, as returned by `get_issue_comments()`.
- **issue**: Optional dataframe or tibble of issues, as returned by `get_issues()`. If provided, output includes issue-level data such as the title, initial description, creation date, etc.
- **link_url**: Boolean. Whether or not to provide link to each item, as provided by `url` column in dataset.

**Details**

HTML output is wrapped in a `<div>` of class `"report_discussion"` for custom CSS styling.

**Value**

Returns character string of HTML with class attribute to be correctly shown "as-is" in RMarkdown.

**See Also**

Other issues: `get_issue_comments`, `get_issue_events`, `get_issues`, `parse_issue_comments`, `parse_issue_events`, `parse_issues`, `post_issue`, `report_progress`, `viz_waterfall`

Other comments: `get_issue_comments`, `parse_issue_comments`
## Examples

```r
## Not run:
# the following could be run in RMarkdown
repo <- create_repo_ref("emilyriederer", "projmgr")
issue <- get_issues(repo, number = 15)
issue_df <- parse_issues(issue)
comments <- get_issue_comments(repo, number = 15)
comments_df <- parse_issue_comments(comments)
report_discussion(issue_df, comments_df)

## End(Not run)
```

---

### report_plan

*Print plan in RMarkdown friendly way*

#### Description

Interprets list representation of plan, using HTML to format results in a highly readable and attractive way. Resulting object returned is a character vector of HTML code with the added class of `"knit_asis"` so that when included in an RMarkdown document knitting to HTML, the results will be correctly rendered as HTML.

#### Usage

```r
report_plan(plan, show_ratio = TRUE)
```

#### Arguments

- **plan**: List of project plan, as returned by `read_plan()`
- **show_ratio**: Boolean. Whether or not to report (# Closed Items / # Total Items) for each group as a ratio

#### Details

The resulting HTML unordered list (`<ul>`) is tagged with class `report_plan` for custom CSS styling.

#### Value

Returns character string of HTML with class attribute to be correctly shown "as-is" in RMarkdown

#### See Also

Other plans and todos: *post_plan, post_todo, read_plan, read_todo, report_todo, template_yaml*
## Examples

```r
# Not run:
# the following could be run in RMarkdown
plan_path <- system.file("extdata", "plan-ex.yml", package = "projmgr", mustWork = TRUE)
my_plan <- read_plan(plan_path)
report_plan(my_plan)

# End(Not run)
```

## Description

Print issue-milestone progress in RMarkdown friendly way

Interprets dataframe or tibble of items (e.g. issues) by breaking apart groups (e.g. milestones), listing each item title as open or closed, and using HTML to format results in a highly readable and attractive way. Resulting object returned is a character vector of HTML code with the added class of 'knit_asis' so that when included in an RMarkdown document knitting to HTML, the results will be correctly rendered as HTML.

## Usage

```r
report_progress(issues, group_var = "milestone_title", link_url = TRUE,
show_ratio = TRUE, show_pct = TRUE)
```

## Arguments

- **issues**: Dataframe or tibble of issues and milestones, as returned by `get_issues()` and `parse_issues()`.
- **group_var**: Character string variable name by which to group issues. Defaults to "milestone_title".
- **link_url**: Boolean. Whether or not to provide link to each item, as provided by `url` column in dataset.
- **show_ratio**: Boolean. Whether or not to report (# Closed Items / # Total Items) for each group as a ratio.
- **show_pct**: Boolean. Whether or not to report (# Closed Items / # Total Items) for each group as a percent.

## Details

The resulting HTML unordered list (\textless ul\textgreater) is tagged with class 'report_progress' for custom CSS styling.

Items without a related group are put into an "Ungrouped" category. Filter these out before using this function if you wish to only show items that are in a group.

## Value

Returns character string of HTML with class attribute to be correctly shown "as-is" in RMarkdown.
See Also

Other issues: get_issue_comments, get_issue_events, get_issues, parse_issue_comments, parse_issue_events, parse_issues, post_issue, report_discussion, viz_waterfall

Examples

```r
## Not run:
repo <- create_repo_ref("emilyriederer", "projmgr")
issues <- get_issues(repo, state = 'all')
issues_df <- parse_issues(issues)
report_progress(issues_df)
## End(Not run)
```

---

**report_taskboard**

Report HTML-based task board of item status

Description

Produces three column task board showing any relevant objects (typically issues or milestones) as "Not Started", "In Progress", or "Done".

Usage

```r
report_taskboard(data, in_progress_when, include_link = FALSE, hover = FALSE, colors = c("#f0e442", "#56b4e9", 
#009e73"))
```

Arguments

data Dataset, such as those representing issues or milestones (i.e. from parse_issues()). Must have state variable.

in_progress_when Function with parameter data that returns Boolean vector. Generally, one of the taskboard helper functions. See ?taskboard_helpers for details.

include_link Boolean whether or not to include links back to GitHub

hover Boolean whether or not tasks should be animated to slightly enlarge on hover

colors Character vector of hex colors for not started, in progress, and complete tasks (respectively)

Details

The following logic is used to determine the status of each issue:

- **Done:** Items with a state of "closed"
- **In Progress:** Custom logic via in_progress_when. See ?taskboard_helpers for details.
- **Not Started:** Default case for items neither In Progress or Closed
**report_todo**

**Value**

Returns character string of HTML/CSS with class attribute to be correctly shown "as-is" in RMarkdown.

**Examples**

```r
## Not run:
# in RMarkdown
```

```
issues <- get_issues(myrepo, milestone = 1)
issues_df <- parse_issues(issues)
report_taskboard(issues_df, in_progress_when = is_labeled_with('in-progress'))
```

````
## End(Not run)
```

---

**Description**

Interprets list representation of to-do list, using HTML to format results in a highly readable and attractive way. Resulting object returned is a character vector of HTML code with the added class of 'knit_asis' so that when included in an RMarkdown document knitting to HTML, the results will be correctly rendered as HTML.

**Usage**

```
report_todo(todo, show_ratio = TRUE)
```

**Arguments**

- `todo`: List of to-do list, as returned by `read_todo()`.
- `show_ratio`: Boolean. Whether or not to report (# Closed Items / # Total Items) for each group as a ratio.

**Details**

The resulting HTML unordered list `<ul>` is tagged with class 'report_todo' for custom CSS styling.

**Value**

Returns character string of HTML with class attribute to be correctly shown "as-is" in RMarkdown.

**See Also**

Other plans and todos: `post_plan`, `post_todo`, `read_plan`, `read_todo`, `report_plan`, `template_yaml`
Examples

```r
## Not run:
# the following could be run in RMarkdown
todo_path <- system.file("extdata", "todo-ex.yml", package = "projmgr", mustWork = TRUE)
my_todo <- read_todo(todo_path)
report_todo(my_todo)

## End(Not run)
```

Description

The `viz_taskboard()` function creates a three-column layout of entities that are not started, in progress, or done. Objects are classified as done when they have a state of "closed". Objects are classified as "To-Do" when they are neither "Closed" or "In Progress". However, what constitutes "In Progress" is user and project dependent. Thus, these functions let users specify what they mean.

Usage

```r
is_labeled()

is_labeled_with(label, any = TRUE)

is_assigned()

is_assigned_to(login, any = TRUE)

is_in_a_milestone()

is_in_milestone(number)

is_created_before(created_date)

is_part_closed()

is_due()

is_due_before(due_date)

has_n_commits(events, n = 1)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>Label name(s) as character vector</td>
</tr>
</tbody>
</table>
When the supplied vector has more than one value, should the result return TRUE if any of those values are present in the dataset (logical OR)

User login(s) as character vector

Milestone number

Date as character in "YYYY-MM-DD" format

Date as character in "YYYY-MM-DD" format

Dataframe containing events for each issue in data

Minimum of commits required to be considered in progress

Details

General options:

• is_created_before: Was created before a user-specified data (as "YYYY-MM-DD" character string)

Issue-specific options:

• is_labeled_with: User-specified label (as character string) exists
• isAssigned: Has been assigned to anyone
• is_assigned_to: Has been assigned to specific user-specified login (as character string)
• is_in_a_milestone: Has been put into any milestone
• is_in_milestone: Has been put into a specific milestone

Milestone-specific options:

• is_part_closed: Has any of its issues closed
• is_due: Has a due date
• is_due_before: Has a due date by or before a user-specified date (as "YYYY-MM-DD" character string)

Value

Function to be passed as in_progress_when argument in viz_taskboard()

Examples

```r
# Not run:
viz_taskboard(issues, in_progress_when = is_labeled_with('in-progress'))
viz_taskboard(milestones, in_progress_when = is_created_before('2018-12-31'))
viz_taskboard(issues, in_progress_when = is_in_milestone())
report_taskboard(issues, in_progress_when = is_labeled_with('in-progress'))
report_taskboard(milestones, in_progress_when = is_created_before('2018-12-31'))
report_taskboard(issues, in_progress_when = is_in_milestone())
```

# End(Not run)
template_yml | Print YAML template to console

**Description**

Prints YAML templates for either a plan or to-do list to the console as an example for developing your own custom plans and to-do lists. Inspired by similar template_ functions included in the pkgdown package.

**Usage**

template_yml(template = c("plan", "todo"))

**Arguments**

template One of "plan" or "todo" denoting template desired

**Details**

Note that depending on the console, text editor, and settings you are using, the template may or may not preserve the necessary whitespace shown in the output. If you copy-paste the template for modification, ensure that it still adheres to traditional YAML indentation.

**Value**

Prints template to console

**See Also**

Other plans and todos: post_plan, post_todo, read_plan, read_todo, report_plan, report_todo

**Examples**

template_yml('plan')
template_yml('todo')

---

viz_gantt | Visualize Gantt-style chart of planned or actual time to completion

**Description**

Produces plot with one vertical bar from the specified start variable's value to the end variable's value. Common uses would be to visualize time-to-completion for issues gotten by (get_issues and parse_issues) or milestones. Bars are colored by duration with longer bars as a darker shade of blue, and start/completion is denoted by points at the ends of the bars.
Usage

```r
viz_gantt(data, start = "created_at", end = "closed_at",
           str_wrap_width = 30)
```

Arguments

- **data**: Dataset, such as those representing issues or milestones (i.e. `parse_issues()` or `parse_milestones()`). Must have unique `title` variable and variables to specify for `start` and `end`.
- **start**: Unquoted variable name denoting issue start date.
- **end**: Unquoted variable name denoting issue end date.
- **str_wrap_width**: Number of characters before text of issue title begins to wrap.

Details

By default, the start date is the issue’s `created_at` date, and the end date is the issue’s `closed_at` date. However, either of these can be altered via the `start` and `end` parameters since these dates might not be reflective of the true timeframe (e.g. if issues are posted well in advance of work beginning.)

Unfinished tasks (where the value of the `end` variable is `NA`) are colored grey and do not have dots on their bars. Unstarted tasks are dropped because user intent is ambiguous in that case.

Value

`ggplot` object

See Also

`viz_linked`

Examples

```r
## Not run:
issues <- get_issues(myrepo, state = "closed")
issues_df <- parse_issues(issues)
viz_gantt(issues_df)
## End(Not run)
```
viz_taskboard

Visualize Agile-style task board of item status

Description

Produces three column task board showing any relevant objects (typically issues or milestones) as ”Not Started”, ”In Progress”, or ”Done”.

Usage

```r
viz_taskboard(data, in_progress_when, str_wrap_width = 30,
              text_size = 3)
```

Arguments

data Dataset, such as those representing issues or milestones (i.e. from `parse_issues()`). Must have state variable.
in_progress_when Function with parameter data that returns Boolean vector. Generally, one of the taskboard helper functions. See `?taskboard_helpers` for details.
str_wrap_width Number of characters before text of issue title begins to wrap
text_size Text size

Details

The following logic is used to determine the status of each issue:

- Done: Items with a state of ”closed”
- In Progress: Custom logic via `in_progress_when`. See `?taskboard_helpers` for details.
- Not Started: Default case for items neither In Progress or Closed

Value

ggplot object

See Also

viz_linked

Examples

```r
## Not run:
issues <- get_issues(myrepo, milestone = 1)
issues_df <- parse_issues(issues)
viz_taskboard(issues_df, in_progress_when = is_labeled_with("in-progress"))
```

```r
viz_taskboard(issues_df, in_progress_when = is_in_a_milestone())
```
**Description**

Creates a four-bar waterfall diagram. Within the specified timeframe, shows initial, newly opened, newly closed, and final open counts. Works with either issues or milestones, as obtained by the `get` and `parse` functions.

**Usage**

```r
viz_waterfall(data, start_date, end_date, start = "created_at", end = "closed_at")
```

**Arguments**

- `data`: Dataset, such as those representing issues or milestones (i.e. `parse_issues()` or `parse_milestones()`). Must have state variable and variables to specify for `start` and `end`.
- `start_date`: Character string in 'YYYY-MM-DD' form for first date to be considered (inclusive).
- `end_date`: Character string in 'YYYY-MM-DD' form for last date to be considered (inclusive).
- `start`: Unquoted variable name denoting issue start date.
- `end`: Unquoted variable name denoting issue end date.

**Details**

The following logic is used to classify issues:

- **Initial**: \(\text{start} < \text{start}\_\text{date} \text{ and (end} > \text{start}\_\text{date or state} == 'open'}\)
- **Open**: \(\text{start} >= \text{start}\_\text{date} \text{ and start} <= \text{end}\_\text{date}\)
- **Closed**: \(\text{end} >= \text{start}\_\text{date} \text{ and end} <= \text{end}\_\text{date}\)
- **Final**: \(\text{start} < \text{end}\_\text{date} \text{ and (end} > \text{end}\_\text{date or state} == 'open'}\)

The exact accuracy of the logic depends on filtering that has already been done to the dataset. Think carefully about the population you wish to represent when getting your data.

**Value**

`ggplot` object

**See Also**

Other issues: `get_issue_comments`, `get_issue_events`, `get_issues`, `parse_issue_comments`, `parse_issue_events`, `parse_issues`, `post_issue`, `report_discussion`, `report_progress`
Examples

```r
## Not run:
viz_waterfall(milestones, '2017-01-01', '2017-03-31')

## End(Not run)
```