# github-collective Documentation

***Release 1.0***

**Plone Collective and contributors**

March 03, 2016

Contents

GitHub organizations are great way for collaborations, companies, and any groups to manage their Git repositories. This tool will let you automate the tedious tasks of creating teams, granting permissions, and creating repositories or modifying their settings. This tool works by utilising the GitHub JSON API to syncronise a local text-based (ini-style) configuration and translate this into configuration for GitHub. This means you can use version control (such as Git) to keep revisions of your configuration and in general, reduce administrative overhead managing your repositories and teams.

Contents:

CHAPTER 1

# Introduction

GitHub organizations are great way for organizations to manage their Git repositories. This tool will let you automate the tedious tasks of creating teams, granting permissions, and creating repositories or modifying their settings.

The approach that the `github-collective` tool takes is that you edit a central configuration (currently an ini-like file) from where options are read and synchronized to GitHub respectively.

Initially, the purpose of this script was to manage Plone's collective organization on GitHub: http://collective.github.com. It is currently in use in several other locations.

CHAPTER 2

# Documentation

Read the full documentation at http://github-collective.rtfd.org.

CHAPTER 3

# Features

- Create one central configuration that you can sync to GitHub to configure your organisation's settings, repositories, teams, and more.
  - Combine this with GitHub's fork-and-pull request model to easily allow non-administrative users to create and manage repositories with minimal overhead.
- Repositories: create and modify repositories within an organization
  - Configure all repository properties as per the GitHub Repos API, including privacy (public/private), description, and other metadata.
  - After the initial repository creation happens, updated values in your configuration will replace those on GitHub.
- Service hooks: add and modify service hooks for repositories.
  - GitHub repositories have support for sending information upon certain events taking place (for instance, pushes being made to a repository or a fork being taken).
  - After the initial repo creation process takes place, updated values in your hook configuration will *replace* those on GitHub.
  - Hooks not present in your configuration (such as those manually added on GitHub or those removed from local configuration) will *not* be deleted.
- Teams: automatically create teams and modify members
  - Control permissions for teams (for example: push, pull or admin)
- Automatically syncs all of the above with GitHub when the tool is run.
- Buildout-style variable substitution in the form `${section:option}`.

CHAPTER 4

# How to install

This package can be installed in a traditional sense or otherwise deployed using Buildout.

## 4.1 Installation

**Tested with** Python2.6

**Dependencies** argparse, requests

```
% pip install github-collective
(or)
% easy_install github-collective
```

## 4.2 Deploy with Buildout

An example configuration for deployment with buildout could look like this:

```
[buildout]
parts = github-collective

[settings]
config = github.cfg
organization = my-organization
admin-user = my-admin-user
password = SECRET
cache = my-organization.cache

[github-collective]
recipe = zc.recipe.egg
initialization = sys.argv.extend('--verbose -C ${settings:cache} -c ${settings:config} -o ${settings
eggs =
    github-collective
```

Deploying in this manner will result in `bin/github-collective` being generated with the relevant options already provided. This means that something calling this script need not provide provide arguments, making its usage easier to manage.

CHAPTER 5

# Usage

When `github-collective` is installed it should create an executable with same name in your *bin* directory.

```
% bin/github-collective --help
usage: github-collective [-h] -c CONFIG [-M MAILER] [-C CACHE] -o GITHUB_ORG
                         -u GITHUB_USERNAME -P GITHUB_PASSWORD [-v] [-p]

This tool will let you automate tedious tasks of creating teams granting
permission and creating repositories.

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        path to configuration file (could also be remote
                        location). eg.
                        http://collective.github.com/permissions.cfg (default:
                        None)
  -M MAILER, --mailer MAILER
                        TODO (default: None)
  -C CACHE, --cache CACHE
                        path to file where to cache results from github.
                        (default: None)
  -o GITHUB_ORG, --github-org GITHUB_ORG
                        github organisation. (default: None)
  -u GITHUB_USERNAME, --github-username GITHUB_USERNAME
                        github account username. (default: None)
  -P GITHUB_PASSWORD, --github-password GITHUB_PASSWORD
                        github account password. (default: None)
  -v, --verbose
  -p, --pretend
```

## 5.1 Locally-stored configuration

```
% bin/github-collective \
    -c example.cfg \ # path to configuration file
    -o vim-addons \  # organization that we are
    -u garbas \      # account that has management right for organization
    -P PASSWORD      # account password
```

## 5.2 Remotely-stored configuration (GitHub)

```
% bin/github-collective \
    -c https://raw.github.com/collective/github-collective/master/example.cfg \
                    # url to configuration file
    -o collective \  # organization that we are
    -u garbas \      # account that has management right for organization
    -P PASSWORD      # account password
```

## 5.3 Cached configuration

```
% bin/github-collective \
    -c https://raw.github.com/collective/github-collective/master/example.cfg \
                    # url to configuration file
    -C .cache       # file where store and read cached results from github
    -o collective \  # organization that we are
    -u garbas \      # account that has management right for organization
    -P PASSWORD      # account password
```

CHAPTER 6

# Configuration

`github-collective` uses a text-based ini-style configuration in typical Python-based style. Essentially, you define a number of sections, each with various options, and the script will parse your configuration and create or update what's on GitHub.

You can consult one of these examples:

- https://raw.github.com/collective/github-collective/master/example.cfg
- http://collective.github.com/permissions.cfg

to get an idea on how to construct your configuration. Read on for specifics regarding the idividual sections and the available options.

**Warning:** For existing GitHub organizations, your configuration should mirror what's on GitHub exactly. If your configuration does not match, this could be destructive. For example, if you have additional repositories on GitHub that aren't in your configuration, they will be removed upon sync as we cannot distinguish whether they are an omission or are missing such that you want them deleted. Run `github-collective` in `pretend` mode first if you're unsure what will happen!

## 6.1 Local Identifiers

If the documentation refers to a *local identifier*, such as that within the `[repo:]` *teams* option, then the given option should contain just the identifier after the colon in the section name being referred to. For example, a section of `[team:my-awesome-team]` would be referenced in the `teams` option as just `my-awesome-team`. If the option in question calls for a list, then each value in the list should follow this.

## 6.2 Variable Substitution

`github-collective` implements Buildout-style variable substitution in the form `${my-section:option}`, which will automatically resolve to the value of `option` within the `[my-section]` section of your configuration.

Here is our example configuration:

```
>>> configuration = """
... [config]
... my-domain = example.org
... my-url = http://${:my-domain}/
...
```

```
... [repo:my-repo]
... owners = ${:main-user}
... homepage = ${config:my-url}
... hooks = travis-ci
... main-user = my-user
... travis-user = ${:owners}
... travis-ci-token = b8cd21c6317a51eeaa752802a0c04454
...
... [hook:travis-ci]
... name = travis
... config =
...     {
...     "user": "${repo:travis-user}",
...     "token": "${repo:travis-ci-token}"
...     }
... events = push
... active = true
... """
```

We can load this configuration to see the result:

```
>>> from githubcollective.config import load_config
>>> from githubcollective.config import substitute, global_substitute
```

```
>>> config = load_config(configuration)
>>> global_substitute(config)
```

Which, after global substitution is applied, will look like the following. Note that there are still some substitutions present - these are *Local* subsitutions and will be resolved in a *context* (in this case a repository context for the given hook options) when the revelant context is being interpreted.

```
>>> from githubcollective.config import output_config
>>> print output_config(config)
[config]
my-domain = example.org
my-url = http://example.org/

[repo:my-repo]
owners = my-user
homepage = http://example.org/
hooks = travis-ci
main-user = my-user
travis-user = my-user
travis-ci-token = b8cd21c6317a51eeaa752802a0c04454

[hook:travis-ci]
name = travis
config =
    {
    "user": "${repo:travis-user}",
    "token": "${repo:travis-ci-token}"
    }
events = push
active = true
```

We can now test our substitution functionality using this configuration as follows. We'll test this by re-initialising the original configuration before it had global subsitution applied.

```
>>> config = load_config(configuration)
```

In the above example, we demonstrate all types of substitution, including substitutions that refer to other substitutions and ensure that these all can be resolved successfully.

### 6.2.1 Global options

These options look like `${config:my-url}` and `${repo:my-repo:hooks-events}`, which refers to a fully-qualified section and option.

For example, using the configuration above, you are able to refer to options like so:

```
>>> substitute('${config:my-domain}', config)
'example.org'
```

```
>>> substitute('${config:my-url}', config)
'http://example.org/'
```

```
>>> substitute('${repo:my-repo:main-user}', config)
'my-user'
```

```
>>> substitute('${hook:travis-ci:name}', config)
'travis'
```

If you attempt to refer to a missing option or section, you'll be informed of this:

```
>>> substitute('${config:idontexist}', config)
...
Traceback (most recent call last):
    ...
NoOptionError: No option 'idontexist' in section: 'config'
```

```
>>> substitute('${idontexist:option}', config)
...
Traceback (most recent call last):
    ...
NoSectionError: No section: 'idontexist'
```

### 6.2.2 Options in same section

Substitution can refer to another option within the same section by omitting the section name like so: `${:main-user}`.

Using the example configuration above, we see we can resolve options with a given context:

```
>>> substitute('${:main-user}', config, context='repo:my-repo')
'my-user'
```

```
>>> substitute('${:events}', config, context='hook:travis-ci')
'push'
```

### 6.2.3 Local options

These are special options that look like `${repo:travis-user}`, which refers to a local option that is resolved at the time relevant section is processed, in the appropriate context. At present, hooks are the only things that belong to

repositories, so attempting to use such a field in anything other than a `[hook:]` context will not work.

For example:

```
>>> substitute('${repo:travis-user}', config,
...            context='repo:my-repo', local=True)
'my-user'
```

```
>>> substitute('${repo:travis-ci-token}', config,
...            context='repo:my-repo', local=True)
'b8cd21c6317a51eeaa752802a0c04454'
```

### 6.2.4 Ordering and options

Options are resolved top-to-bottom within the configuration, with the exception of *Local* options that are resolved when instantiated (for instance, when the hook for a repo is created, as hooks exist per-repsository). So, in the example above, the parser will consider all options in `[repo:my-repo]` in the order they were defined, and then when adding `[hook:travis-ci]` to the repository, *Local* options will be resolved in the context of said repository. Doing so means you are able to have one common hook configuration, but have *per-repository* configuration options, such as those for Travis-CI tokens, passwords, URLs, and more.

Keep in mind that there are no restrictions on arbitrary section names so your variable storage can be unbounded. This also means you could conceivably utilise the same configuration file for multiple purposes (such as for `github-collective` and a Paster application) and share variables.

Substitution will attempt to alert you of circular dependencies and provide some explaination why a substitution is failing in the form of a raised Python exception with suitable details.

```
>>> broken_config = """
... [config]
... my-domain = ${:my-url}
... my-location = ${:my-domain}
... my-url = ${:my-location}
... """
```

```
>>> broken = load_config(broken_config)
>>> global_substitute(broken)
...
Traceback (most recent call last):
    ...
ValueError: Circular reference in substitutions ${:my-url} --> ${:my-location} --> ${:my-domain} -->
```

## 6.3 Repositories

Repositories form the basis for your code hosting on GitHub. Using a `[repo:]` section within your configuration, the script will automatically create a new repository with the relevant settings, or update a repository if it already exists. Alternatively, you can specify to fork an existing repository as well.

### 6.3.1 Examples

Keep in mind that all of the options given are not always required but are set out here to demonstrate what you can do.

We can create a new repository, using various options allowable by the GitHub Repos API:

```
[repo:collective.demo]
owners = davidjb
teams = contributors
hooks =
    my-jenkins
    some-website
description = My awesome repo
homepage = http://example.org
has_issues = false
has_wiki = false
has_downloads = false
```

As the example suggests, this will create a repository with the name of `collective.demo`, assign `davidjb` administrative rights and the `contributors` team push and pull rights, and create the relevant service hooks. The repository will the given metadata applied to it and options set. If we later go and change the above configuration (or indeed if the repository already exists on GitHub), then differences will be synced to GitHub. For instance, we could change `has_issues` to `true` to enable the issue tracker again, add or remove `hooks`, and more.

We can also fork a repository that already exists:

```
[repo:github-collective]
fork = collective/github-collective
owners = garbas
```

Finally, in a special example, we can create a repository as `Private`, if you are using `github-collective` against a paid-for GitHub organization like so:

```
[repo:collective.demo]
owners = davidjb
private = true
```

This will fail if your GitHub organization lacks sufficient quota (for instance, those that are free only).

### 6.3.2 Section configuration

When creating or updating a repository, arbitrary options provided within a `[repo:]` section will be sent as part of the relevant POST request. For all potential options, see the GitHub Repos API documentation. All values are optional (with the exception of `name`, which must be specified already in our configuration) and GitHub provides defaults for many of the options as per the documentation. Note that values that GitHub expects as Boolean (for example `private`, `has_issues` and so forth) will be coerced accordingly as per standard Python ini-syntax.

There are special options, however, which are not sent but rather used locally in configuring a repository. These are:

***owners* (optional)** List of GitHub user names to set as *Owners* of a repository. Within GitHub's interface, these users are seen to possess the *Push, Pull & Administrative* permission. This should not be confused with Owners of an entire GitHub organization.

***teams* (optional)** List of local string identifiers for collaborators of a repository. Teams specified here will be granted the appropriate permission to the given repository (see Teams configuration). The identifiers in this option should refer to relevant `[team:]` sections in the local configuration. This option is the inverse of `repos` for repository configuration.

***hooks* (optional)** List of string identifiers for GitHub service hooks, referring to relevant `[hook:]` sections in the local configuration. This list should contain just the identifier after the colon in the section name. For example, a section of `[hook:my-webhook]` would be referenced in the `hooks` option as just `my-webhook`. Service hooks specified here will be either created or updated against the repository.

Forking is a special case and settings in your configuration will not be sent to GitHub until updating the repository takes place.

## 6.4 Teams

Groups of users on GitHub organizations can be set out into Teams. Using `[team:]` sections, you can create as many teams as you'd like and assign them access to repositories. You can achieve this by either assigning repositories to teams, or teams to repositories - they are both equivalent.

### 6.4.1 Examples

In order to create a Team of users with the ability to push and pull from certain repositories, the configure would look like:

```
[team:contributors]
permission = push
members =
    MarcWeber
    honza
    garbas
repos =
    snipmate-snippets
    ...

[repo:snipmate-snippets]
    ...
```

Similarly, we can achieve the same with inverting the `repos` option into `teams` on the repository configuration:

```
[team:contributors]
permission = push
members =
    MarcWeber
    honza
    garbas

[repo:snipmate-snippets]
teams =
    contributors
```

By changing the `permission` option, you will affect what the users of that Team can do on the repositories they're assigned to. See below for details.

### 6.4.2 Section configuration

Each `[team:]` section within your configuration can utilise the following values.

***permission* (optional)** The permission to assign to this group. At time of writing, GitHub has three types of permissions available for Teams:

- `push`: team members can pull, but not push to or administer repositories.
- `pull`: team members can pull and push, but not administer repositories.
- `admin`: team members can pull, push and administer repositories.

If not provided, this option defaults to `pull`.

***members* (optional)** List of GitHub user names to set as part of this Team. These users will be granted the `permission` above to any repositories this Team is configured against.

***repos* (optional)** List of string identifiers of repositories this Team should have the given permission against. The identifiers in this option should refer to relevant `[repo:]` sections in the local configuration. This option is the inverse of `teams` for repository configuration.

## 6.5 Service hooks

GitHub allows repositories to be configured with *service hooks*, which allow GitHub to communicate with a web server (and thus web services) when certain actions take place within that repository. These can be configured via GitHub's web interface through the `Admin` page for repositories, in the `Service Hooks` section, which provides most options, or else via GitHub's API, which provides some additional hidden settings.

For an introduction to this topic, consult the Post-Receive Hooks documentation.

Effectively, GitHub will send a POST request to a given web-based endpoint with relevant information about commits and metadata about the repository when a certain trigger happens. The GitHub Hooks API has complete details about what event triggers are available, details about what services are available, and more.

### 6.5.1 Examples

As a worked example, you can configure a repository you have to send details about commits and changes as they happen to a Jenkins CI instance in order for continuous testing to take place. You would enter the following in your `github-collective` configuration like so:

```
[hook:my-jenkins-hook]
name = web
config =
    {"url": "https://jenkins.plone.org/github-webhook/",
    "insecure_ssl": "1"
    }
active = true

[repo:collective.github.com]
...
hooks =
    my-jenkins-hook
```

The result here is that, once run, the `collective.github.com` repository will have a `web` hook created against it that instructs GitHub to send the relevant POST payload to the given `url` in question. This hook creation is effectively synonymous with adding a hook via the web-based interface, with the one minor exception in that we provide an extra value for `insecure_ssl` to ensure that GitHub will communicate with our non-CA signed certificate.

Our `[repo:]` section has a `hooks` option in which you can specify the identifiers of one or more hooks within your configuration. This option is not required, however, should you have no service hooks.

See the next section for specifics and how to configure these types of sections within your `github-collective` configuration.

### 6.5.2 Section configuration

Each `[hook:]` section within your configuration can utilise the following values. Options provided here will be coerced from standard ini-style options into suitable values for posting JSON to GitHub's API. For specifications, refer to https://api.github.com/hooks

***name* (required)** String identifier for a service hook. Refer to specification for available service identifiers or to the Service Hooks administration page for your repository. One of the most commonly used options is `web` for generic web hooks (seen as *Brook URLs* in the Service Hooks administration page).

***config* (required)** Valid JSON consisting of key/value pairs relating to configuration of this service. Refer to specifications for applicable config for each service type.

*Note*: if a change is made to your local configuration, `github-collective` will attempt to update hook settings on GitHub. If you have Boolean values present in this option, then in order to prevent `github-collective` from attempting to update GitHub on every run, these values should exist as strings - either `"1"` or‘“”0”‘‘ - as this is how GitHub stores configuration (and we compare against this to check whether we need to sync changes).

***events* (optional)** List of events the hook should apply to. Different services can respond to different events. If not provided, the hook will default to `push`. Keep in mind that certain services only listen for certain types of events. Refer to API specification for information.

***active* (optional)** Boolean value of whether the hook is enabled or not.

CHAPTER 7

# Gotchas

- URLs specified within the configuration should possess a trailing slash where appropriate, for instance `http://example.com` (no trailing slash) will, when returned by GitHub, become `http://example.com/`. This means that your configuration files will appear out of sync and thus `github-collective` will attempt to update every run.
- Boolean values stored within JSON Hook configuration should be either `0` or `1` and strings, as this is what GitHub stores. Read the section on *Service hooks* for more information.

CHAPTER 8

# Testing

nose is utilised for testing and configuration for nose exists within the setup.cfg file within this project. This configuration automatically examines files for tests within the project, including this read-me itself. You can initialise and run tests using the Buildout configuration provided:

```
git clone git://github.com/collective/github-collective.git
cd github-collective
virtualenv .
python boostrap.py
bin/buildout
bin/nosetests
```

tox is used to ensure this package installs correctly under each version of Python. Currently we test Python 2.6 and Python 2.7. Support for running tests under tox will come shortly. To test installation:

```
git clone git://github.com/collective/github-collective.git
cd github-collective
virtualenv .
pip install tox
tox
```

CHAPTER 9

# Issues and Contributing

Report issues via this project's GitHub issue tracker at https://github.com/collective/github-collective/issues.

Contribute by submitting a pull request on GitHub or else by adding yourself to the Collective and contributing directly.

CHAPTER 10

# Todo

- Allow configuration of organisation settings via API
- Add facility to continue if error experienced
- Send emails to owners about removing repos
- Better logging mechanism (eg. logbook)
- Support configuration extensibility (eg `extends =` syntax) for using multiple configuration files.

CHAPTER 11

# Credits

**Author** Rok Garbas (garbas)

**Contributor** David Beitey (davidjb)

CHAPTER 12

# Indices and tables

- genindex
- modindex
- search