RepoBee Documentation

Release 2.3.1

Simon Larsén

Oct 06, 2019

Contents:

If you are new to RepoBee, the *Introduction* and *RepoBee User Guide* sections are must-reads. Developers looking to work on RepoBee, or fork it, are probably most interested in the modindex. Developers looking to create plugins should head over to the documentation for repobee-plug.

If you use the *RepoBee User Guide* in any way and feel like skipping *Getting started (the show-config, verify-settings and setup commands)*, make sure to read *Configure RepoBee for the target organization (show-config and verify-settings)* anyway! The rest of the guide assumes a configuration as described there.

Important: Please open an issue over on the issue tracker if you find documentation bugs, have trouble understanding something or think something is missing. Especially when it comes to the userguide, which is intended to be as intuitive as possible, please do provide feedback if you get stuck.

CHAPTER 1

Introduction

RepoBee is an opinionated tool for managing anything from a handful to thousands of Git repositories on the GitHub and GitLab platforms. There were two primary reasons for RepoBee's inception. First, the old teachers_pet tool that we used previously lacked some functionality that we needed and had been archived in favor of the new GitHub Classroom. Second, GitHub Classroom did not support GitHub Enterprise at the time (and as of this writing, still does not, although efforts have been made to that end). RepoBee is heavily inspired by teachers_pet, as we enjoyed the overall workflow, but improves on the user experience.

1.1 Philosophy and goals

When RepoBee was first created, it's goals were simple: facilitate creation and management of student repositories for courses at KTH, using GitHub Enterprise. Since then, the scope of the project has grown to incorporate many new features, including support for the GitLab platform. For new users of Git/GitHub/GitLab in education, RepoBee provides both a tool to make it happen, and an opinionated workflow to ease the transition. For the more experienced user, the plugin system can be used to extend or modify the behavior of RepoBee. While creating a plugin requires some rudimentary skills with Python programming, installing a plugin created by someone else is no harder than installing RepoBee itself. The plugin system enables RepoBee to both be easy to get up and running *without* any required customization, while still *allowing* for a degree of customization to those that want it. See *Plugins for RepoBee* for details.

Another key goal is to keep RepoBee simple to use and simple to maintain. RepoBee requires a minimal amount of static data to operate (such as a list of students, a URL to the platform instance and an access token to said platform), which can all be provided in configuration files or on the command line, but it does not require any kind of backing database to keep track of repositories. That is because RepoBee itself does not keep track of anything, except possibly for the aforementioned static data if one chooses to keep it in configuration files. All of the complex state state is more or less implicitly stored on the hosting platform, and RepoBee locates student repositories based on strict naming conventions that are adhered to by all of its commands. This allows RepoBee to be simple to set up and use on multiple machines, which is crucial in a course where multiple teachers and TAs are managing the student repositories. RepoBee is very intentionally designed *not* to be a service, but an on-demand tool that is only in use when explicitly being used. This means that nothing needs to be installed server-side for RepoBee to function, which also happens to be key to supporting multiple hosting platforms. For an experienced user, installing RepoBee and setting everything

up for a new course can literally take minutes. For the novice, the *RepoBee User Guide* will hopefully prove sufficient to get started within the hour.

1.2 Key concepts

Some terms occur frequently in RepoBee and are best defined up front. Some of the descriptions may not click entirely before reading the *RepoBee User Guide* section, so quickly browsing through these definitions and re-visiting them when needed is probably the best course of action. As GitHub is the default platform, these concepts are based on and often refer to GitHub-specific terms. For a mapping to GitLab terms and concepts, please see the *RepoBee and GitLab* section.

- *Platform*: Or *hosting platform*, refers to services such as GitHub and GitLab.
- *Platform instance:* A specific instance of a hosting platform. For example, https://github.com is one instance, and an arbitrary GitLab Enterprise installation is another.
- *Target organization*: The GitHub Organization related to the current course round.
- *Master repository*: Or *master repo*, is a template repository upon which student repositories are based.
- *Master organization*: The master organization is an optional organization to keep master repos in. The idea is to be able to have the master repos in this organization to avoid having to migrate them to the target organization for each course round. It is highly recommended to use a master organization if master repos are being worked on across course rounds.
- *Student repository*: Or *student repo*, refers to a *copy* of a master repo for some specific student or group of students.

1.3 Conventions

The following conventions are fundamental to working with RepoBee.

- For each course and course round, use one target organization.
- Any user of RepoBee has unrestricted access to the target organization (i.e. is an owner). If the user has limited access, some features may work, while others may not.
- Master repos should be available as private repos in one of three places:
  - The master organization (recommended if the master repos are being maintained and improved across course rounds).
  - The target organization. If you are doing a trial run or for some reason can't have multiple organizations, this may be a good option.
  - Locally in the current working directory. If your master repos are trivial (e.g. empty), this may be a good option.
- Student repositories are copies of the default branches of the master repositories (i.e. `--single-branch` cloning is used by default). That is, until students make modifications.
- Student repositories are named *<username>-<master_repo_name>* to guarantee unique repo names. - Student repositories belonging to groups of students are named *<username-1>-<username-2>-...-<master-repo-name>*.
- Each student is assigned to a team with the same name as the student's username (or a concatenation of usernames for groups). It is the team that is granted access to the repositories, not the student's actual user.

- Student teams have push access to the repositories, but not administrative access (i.e. students can't delete their own repos).

Note: RepoBee has no way of enforcing these conventions, other than itself strictly adhering to them. For example, there are no countermeasures against someone manually changing the names of student repositories or their URLs, and as there are endless variations of things that can be manually changed, there are no safety checks against such things either. If you have a need to manually change something, do keep in mind that straying from RepoBee's conventions may cause it to act unexpectedly.

1.4 Usage with different platforms (GitHub, GitHub Enterprise and GitLab)

RepoBee was originally designed for use with GitHub Enterprise, but also works well with the public cloud service at https://github.com. Usage of RepoBee should be identical, but there are two differences between the two that one should be aware of.

Note: As of v1.6.0, GitLab is supported by most features. Please see *RepoBee and GitLab* for more information on which commands work, and how to use RepoBee with GitLab.

1.4.1 The Organization must have support for private repositories

Private repositories are key to keep students from being able to see each others' work, and thereby avoid a few avenues for plagiarism.

- **Enterprise:** All Organizations on Enterprise support private repositories.
- **github.com:** You need a paid Organization (confusingly called a *Team*, but unrelated to the Teams *inside* an Organization). Educators and researchers can get such Organization accounts for free, see how to get the discount here.
- **GitLab:** All GitLab groups (self-hosted and on https://gitlab.com) support private repositories.

1.4.2 Students are added to the target Organization slightly differently

During setup, students are added to their respective Teams. Precisely how this happens differs slightly.

- **Enterprise:** Students are automatically added to their Teams in the Organization.
- **github.com:** Students are invited to the Organization and added to their Teams upon accepting.
- **GitLab:** Students are automatically added, both on self-hosted and https://gitlab.com.

CHAPTER 2

Install

2.1 Requirements

RepoBee requires Python 3.5+ and a somewhat up-to-date version of Git (2.0+ to be on the safe side). Officially supported operating systems are Ubuntu 17.04+ and macOS, but RepoBee should run fine on any Linux distribution and also on WSL on Windows 10. Please report any issues with operating systems and/or Git versions on the issue tracker.

2.2 Check your Python version

For RepoBee to run, you need to have Python 3.5 or later. On many operating systems, `python` is an alias for Python 2.7, and `python3` is an alias for the latest version of Python 3 that is installed. For this install guide, python3 is assumed to be a Python version 3.5 or higher. You can check the version yourself with:

```
$ python3 --version
# or
$ python --version
```

Then, just use whichever of those Pythons claim to be 3.5 or higher.

2.3 Option 1: Install from PyPi with *pip*

The latest release of RepoBee is on PyPi, and can thus be installed as usual with pip. I strongly discourage system-wide pip installs (e.g. `sudo pip install <package>`), as this may land you with incompatible packages in a very short amount of time. A per-user install can be done like this:

1. Execute `python3 -m pip install --user --upgrade repobee` to install the package.
2. **Run `repobee -h` to verify that you can find the script.**

- If that doesn't work, the `repobee` script can't be found on your PATH variable. Try `python3 -m repobee -h` to run the main module of RepoBee, which is equivalent to `repobee -h`.

This same install command should also be good for upgrading RepoBee to a new version.

Important: Of course, if `python` corresponds to Python 3 on your system, use that instead of `python3` in the command shown above.

Important: A `--user` install will perform a local install for the current user. Any scripts will be installed in a user-local bin directory. If this directory is not on your path (which it often is not by default), you will not be able to run the `repobee` script (however, `python -m repobee` should still work). pip should issue a warning about this, including the path to the local bin directory. To resolve the problem, add the local bin directory to your $PATH variable. When installing, pip will usually complain that the bin directory is not on the $PATH variable and point out where the directory is located.

2.4 Option 2: Clone the repo and the install with *pip*

If you want the dev version, you will need to clone the repo, as only release versions are uploaded to PyPi. Unless you are planning to work on this yourself, I suggest going with the release version.

1. **Clone the repo with git:**
    - `git clone https://github.com/repobee/repobee`
2. `cd` into the project root directory with `cd repobee`.
3. **Install locally with `pip`.**
    - `python3 -m pip install --user --upgrade .`, this will create a local install for the current user.
    - Or just `pip install .` if you use `virtualenv`.
    - For development, use `pip install -e .[TEST]` in a `virtualenv`.

CHAPTER 3

RepoBee User Guide

3.1 Getting started (the `show-config`, `verify-settings` and `setup` commands)

Important: This guide assumes that the user has access to a `bash` shell, or is tech-savvy enough to translate the instructions into some other shell environment.

Important: Whenever you see specific mentions of GitHub, refer to the *RepoBee and GitLab* section for how this translates to use with GitLab.

The basic workflow of RepoBee is best described by example. In this section, I will walk you through how to set up a target organization with master and student repositories by showing every single step I would perform myself. The basic workflow can be summarized in the following steps:

1. Create an organization (the target organization).
2. Configure RepoBee for the target organization.
3. Verify settings.
4. Set up the master repos.
5. Set up the student repos.

This should leave you with enough knowledge to use the rudimentary features of RepoBee. There is much more to RepoBee, such as opening/closing issues, updating student repos and cloning repos in batches. This is covered in later sections, but you don't necessarily need to go through the entire guide in one go. Now, let's delve into the above steps in greater detail.

3.1.1 Create an organization

This is an absolutely necessary pre-requisite for using RepoBee. Create an organization with an appropriate name on the platform instance you intend to use. You can find the `New organization` button by going to `Settings -> Organization`. I will call my *target organization* `repobee-demo`, so whenever you see that, substitute in the name of your target organization.

Important: At KTH, we most often do not want our students to be able to see each others' repos. By default, however, members have read access to *all* repos. To change this, go to the organization dashboard and find your way to `Settings -> Member privileges`. There should be a drop-down called something along the lines of "Base permissions" or "Default repository settings", which you will want to set to `None`. The placement and name of this drop-down has changed places twice since the first iteration of this documentation, so it may not be an exact match, but you should find it somewhere around there.

3.1.2 Configure RepoBee for the target organization (`show-config` and `verify-settings`)

For the tool to work at all, it needs to be provided with an access token to whichever platform instance you intend to use. See the GitHub access token docs for how to create a token. The token should have the `repo` and `admin:org` scopes.

Note: See *Getting an access token for GitLab* if you use GitLab!

While you can set this token in an environment variable (see *Configuration*), it's more convenient to just put it in the configuration file, as you will put other default values in there. The `config-wizard` command starts a configuration wizard that prompts you for default values for the available settings. The defaults that are set in the configuration file are *just defaults*, and can always be overridden on the command line. For the rest of this guide, I will assume that the config file has defaults for at least the following:

Listing 1: config.cnf

```
[DEFAULTS]
base_url = https://some-enterprise-host/api/v3
user = slarse
org_name = repobee-demo
master_org_name = master-repos
token = SUPER_SECRET_TOKEN
```

Now, run `repobee config-wizard` and enter your own values for the options shown above. To skip an option, simply press ENTER without first typing in a value. Here are some pointers regarding the different values:

- **Enter the correct url for your platform instance. There are two options:**
  - If you are working with GitHub Enterprise, simply replace `some-enterprise-host` with the appropriate hostname.
  - If you are working with `github.com`, replace the whole url with `https://api.github.com`.
- Replace `slarse` with your GitHub username.
- Replace `repobee-demo` with whatever you named your target organization.
- Replace `SUPER_SECRET_TOKEN` with your access token.
- **Replace `master_org_name` with the name of the organization with your master repos.**

  - It you keep the master repos in the target organization or locally, **skip this option**.

- **If you are using GitLab:**
  - The `base_url` should be to the host, not to the API endpoint. I.e. if you are using https://gitlab.com, then the `base_url` option should simply read `https://gitlab.com`.
  - Enter `gitlab` for the `plugins` option.

That's it for configuration. The `show-config` command can be used to check that you got everything correctly.

```
$ repobee show-config
[INFO] Found valid config file at /home/slarse/.config/repobee/config.cnf
[INFO]
----------------BEGIN CONFIG FILE-----------------
[DEFAULTS]
base_url = https://some-enterprise-host/api/v3
user = slarse
org_name = repobee-demo
master_org_name = master-repos
token = SUPER_SECRET_TOKEN
-----------------END CONFIG FILE------------------
```

If you ever want to re-configure some of the options, simply run the `config-wizard` command again.

3.1.3 Verify settings

Important: `verify-settings` is not yet supported by the `gitlab` plugin.

Now that everything is set up, it's time to verify all of the settings. Given that you have a configuration file that looks something like the one above, you can simply run the `verify-settings` command without any options.

```
$ repobee verify-settings
[INFO] Verifying settings ...
[INFO] Trying to fetch user information ...
[INFO] SUCCESS: found user slarse, user exists and base url looks okay
[INFO] Verifying access token scopes ...
[INFO] SUCCESS: access token scopes look okay
[INFO] Trying to fetch organization ...
[INFO] SUCCESS: found organization test-tools
[INFO] Verifying that user slarse is an owner of organization repobee-demo
[INFO] SUCCESS: user slarse is an owner of organization repobee-demo
[INFO] Trying to fetch organization master-repos ...
[INFO] SUCCESS: found organization master-repos
[INFO] Verifying that user slarse is an owner of organization master-repos
[INFO] SUCCESS: user slarse is an owner of organization master-repos
[INFO] GREAT SUCCESS: All settings check out!
```

If any of the checks fail, you should be provided with a semi-helpful error message. When all checks pass and you get `GREAT SUCCESS`, move on to the next section!

3.1.4 Set up master repos

How you do this will depend on where you want to have your master repos. I recommend having a separate, persistent organization so that you can work on repos across course rounds. If you already have a master organization with your

master repos set up somewhere, and `master_org_name` is specified in the config, you're good to go. If you need to migrate repos into the target organization (e.g. if you keep master repos in the target organization), see the *Migrate repositories into the target (or master) organization (migrate command)* section. For all commands but the `migrate` command, the way you set this up does not matter as far as RepoBee commands go.

Note: Recall that there is nothing special about master repos, they are just your templates for student repos. If you have an organization set up with template repositories, then that is a viable master organization.

3.1.5 Set up student repositories

Now that the master repos are set up, it's time to create the student repos. While student usernames *can* be specified on the command line, it's often convenient to have them written down in a file instead. Let's pretend I have three students with usernames `slarse`, `glassey` and `glennol`. I'll simply create a file called `students.txt` and type each username on a separate line.

Listing 2: students.txt

```
slarse
glassey
glennol
```

Note: **Since v1.3.0:** It is now possible to specify groups of students to get access to the same repos by putting multiple usernames on the same line, separated by spaces. For example, the following file will put *slarse* and *glassey* in the same group.

```
slarse glassey
glennol
```

See *Group assignments* for details.

An absolute file path to this file can be added to the config file with the `students_file` option (see *Configuration file*). Since I often manage different sets of students, that's seldom convenient for me, but if you always manage the same set of students I recommend setting that option so you can omit it from the command line arguments. Now, I want to create one student repo for each master repo and student. The repo names will be on the form `<username>-<master-repo-name>`, guaranteeing their uniqueness. Each student will also be added to a team (which bears the same name as the student's user), and it is the team that is allowed access to the student's repos, not the student's actual user. That all sounded fairly complex, but again, it's as simple as issuing a single command with RepoBee.

```
$ repobee setup --mn task-1 task-2 --sf students.txt
[INFO] Cloning into master repos ...
[INFO] Cloning into file:///home/slarse/tmp/task-1
[INFO] Cloning into file:///home/slarse/tmp/task-2
[INFO] Created team glennol
[INFO] Created team glassey
[INFO] Created team slarse
[INFO] Adding members glennol to team glennol
[WARNING] user glennol does not exist
[INFO] Adding members glassey to team glassey
[INFO] Adding members slarse to team slarse
[INFO] Creating student repos ...
[INFO] Created repobee-demo/glennol-task-1
```

(continues on next page)

(continued from previous page)

```
[INFO] Created repobee-demo/glassey-task-1
[INFO] Created repobee-demo/slarse-task-1
[INFO] Created repobee-demo/glennol-task-2
[INFO] Created repobee-demo/glassey-task-2
[INFO] Created repobee-demo/slarse-task-2
[INFO] Pushing files to student repos ...
[INFO] Pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glassey-task-2 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glassey-task-1 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/slarse-task-1 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glennol-task-2 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glennol-task-1 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/slarse-task-2 master
```

Note: If you have specified the `students_file` option in the config file, then you don't need to specify `--sf students.txt` on the command line. Remember also that options specified on the command line always take precedence over those in the configuration file, so you can override the default students file if you wish by specifying `--sf`..

Note that there was a `[WARNING]` message for the username `glennol`: the user does not exist. At KTH, this is common, as many (sometimes most) first-time students will not have created their GitHub accounts until sometime after the course starts. These students will still have their repos created, but the users need to be added to their teams at a later time (to do this, simply run the `setup` command again for these students, once they have created accounts). This is one reason why we use teams for access privileges: it's easy to set everything up even when the students have yet to create their accounts (given that their usernames are pre-determined).

And that's it for setting up the course, the organization is primed and the students should have access to their repositories!

3.2 Updating student repositories (the `update` command)

Sometimes, we find ourselves in situations where it is necessary to push updates to student repositories after they have been published. As long as students have not started working on their repos, this is fairly simple: just push the new files to all of the related student repos. However, if students have started working on their repos, then we have a problem. Let's start out with the easy case where no students have worked on their repos.

3.2.1 Scenario 1: Repos are unchanged

Let's say that we've updated `task-1`, and that users `slarse`, `glassey` and `glennol` should get the updates. Then, we simply run `update` like this:

```
$ repobee update --mn task-1 -s slarse glennol glassey
[INFO] Cloning into master repos ...
[INFO] Cloning into https://some-enterprise-host/repobee-demo/task-1
[INFO] Pushing files to student repos ...
[INFO] Pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/slarse-task-1 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glennol-task-1 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glassey-task-1 master
[INFO] Done!
```

That's all there is to it for this super simple case. But what if `glassey` had started working on `glassey-task-1`?

Note: Here, `-s slarse glennol glassey` was used to directly specify student usernames on the command line, instead of pointing to a students file with `--sf students.txt`. All commands that require you to specify student usernames can be used with either the `-s|--students` or the `--sf|--students-file` options.

3.2.2 Scenario 2: At least 1 repo altered

Let's assume now that `glassey` has started working on the repo. Since we do not force pushes to the student repos, the push to `glassey-task-1` will be rejected. This is good, we don't want to overwrite a student's progress because we messed up with the original repository. There are a number of things one *could* do in this situation, but in RepoBee, we opted for a very simple solution: open an issue in the student's repo that explains the situation.

Important: If you don't specify an issue to `repobee update`, rejected pushes will simply be ignored.

So, let's first create that issue. It should be a Markdown-formatted file, and the **first line in the file will be used as the title**. Here's an example file called `issue.md`.

Listing 3: issue.md

```
This is a nice title

### Sorry, we messed up!
There are some grave issues with your repo, and since you've pushed to the
repo, you need to apply these patches yourself.

<EXPLAIN CHANGES>
```

Something like that. If the students have used `git` for a while, it may be enough to include the ouptut from `git diff`, but for less experienced students, plain text is more helpful. Now it's just a matter of using `repobee update` and including `issue.md` with the `-i|--issue` argument.

```
$ repobee update --mn task-1 -s slarse glennol glassey -i issue.md
[INFO] Cloning into master repos ...
[INFO] Cloning into https://some-enterprise-host/repobee-demo/task-1
[INFO] Pushing files to student repos ...
[INFO] Pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/slarse-task-1 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glennol-task-1 master
[ERROR] Failed to push to https://some-enterprise-host/repobee-demo/glassey-task-1
return code: 128
fatal: repository 'https://some-enterprise-host/repobee-demo/glassey-task-1/' not␣
↪found
[WARNING] 1 pushes failed ...
[INFO] Pushing, attempt 2/3
[ERROR] Failed to push to https://some-enterprise-host/repobee-demo/glassey-task-1
return code: 128
fatal: repository 'https://some-enterprise-host/repobee-demo/glassey-task-1/' not␣
↪found
[WARNING] 1 pushes failed ...
[INFO] Pushing, attempt 3/3
[ERROR] Failed to push to https://some-enterprise-host/repobee-demo/glassey-task-1
return code: 128
```

(continues on next page)

(continued from previous page)

```
fatal: repository 'https://some-enterprise-host/repobee-demo/glassey-task-1/' not␣
↪found
[WARNING] 1 pushes failed ...
[INFO] Opening issue in repos to which push failed
[INFO] Opened issue glassey-task-1/#1-'Nice title'
[INFO] Done!
```

Note that RepoBee tries to push 3 times before finally giving up and opening an issue, as a failed push could be due to any number of reasons, such as connection issues and misaligned planets.

Note: If you forget to specify the `-i|--issue` argument and get a rejection, you may simply rerun `update` and add it. All updated repos will simply be listed as `up-to-date` (which is a successful update!), and the rejecting repos will still reject the push. However, be careful not to run `update` with `-i` multiple times, as it will then open multiple issues.

3.3 Opening and Closing issues (the `open-issues` and `close-issues` commands)

Sometimes, the best way to handle an error in a repo is to simply notify affected students about it. This is especially true if the due date for the assignment is rapidly approaching, and most students have already started modifying their repositories. There can also be cases where you want to make general announcements, or communicate some other action item that's best highly related to the code that the students are writing. Therefore, RepoBee provides the `open-issues` command, which can open issues in bulk. When the time is right, issues can be closed with the `close-issues` command. Finally, `list-issues` provides a way of quickly seeing what issues are open and closed in student repositories.

3.3.1 Opening Issues

The `open-issues` command is very simple. Before we use it, however, we need to write a Markdown-formatted issue. Just like with the `update` command, the **first line of the file is the title**. Here is `issue.md`:

Listing 4: issue.md

```
An important announcement

### Dear students
I have this important announcement to make.

Regards,
_The Announcer_
```

Awesome, that's an excellent issue. Let's open it in the `task-2` repo for our dear students `slarse`, `glennol` and `glassey`, who are listed in the `students.txt` file (see *Set up student repositories*).

```
$ repobee open-issues --mn task-2 --sf students.txt -i issue.md
[INFO] Opened issue slarse-task-2/#1-'An important announcement'
[INFO] Opened issue glennol-task-2/#1-'An important announcement'
[INFO] Opened issue glassey-task-2/#1-'An important announcement'
```

From the output, we can read that in each of the repos, an issue with the title `An important announcement` was opened as issue nr 1 (`#1`). The number isn't that important, it's mostly good to note that the title was fetched correctly. And that's it! Neat, right?

3.3.2 Closing Issues

Now that the deadline has passed for `task-2`, we want to close the issues opened in *open*. The `close-issues` command takes a *regex* that runs against titles. All issues with matching titles are closed. While you *can* make this really difficult, closing all issues with the title `An important announcement` is simple: we provide the regex `\AAn important announcement\Z`.

```
$ repobee close-issues --mn task-2 --sf students.txt -r '\AAn important announcement\Z
↪'
[INFO] Closed issue slarse-task-2/#1-'An important announcement'
[INFO] Closed issue glennol-task-2/#1-'An important announcement'
[INFO] Closed issue glassey-task-2/#1-'An important announcement'
```

And there we go, easy as pie!

Note: Enclosing a regex expression in `\A` and `\Z` means that it must match from the start of the string to the end of the string. So, the regex used here *will* match the title `An important announcement`, but it will *not* match e.g. `An important anouncement and lunch` or `Hey An important announcement`. In other words, it matches exactly the title `An important announcement`, and nothing else. Not even an extra space or linebreak is allowed.

3.3.3 Listing Issues

It can often be interesting to check what issues exist in a set of repos, especially so if you're a teaching assistant who just doesn't want to leave your trusty terminal. This is where the `list-issues` command comes into play. Typically, we are only interested in open issues, and can then use list issues like so:

```
$ repobee list-issues --mn task-2 --sf students.txt
[INFO] slarse-task-2/#1:  Grading Criteria created 2018-09-12 18:20:56 by glassey
[INFO] glennol-task-2/#1:  Grading Criteria created 2018-09-12 18:20:56 by glassey
[INFO] glassey-task-2/#1:   Grading Criteria created 2018-09-12 18:20:56 by glassey
```

So, just grading critera issues posted by the user `glassey`. What happened to the important announcements? Well, they are closed. If we want to se closed issues, we must specifically say so with the `--closed` argument.

```
$ repobee list-issues --mn task-2 --sf students.txt --closed
[INFO] slarse-task-2/#2:  An important announcement created 2018-09-17 17:46:43 by␣
↪slarse
[INFO] glennol-task-2/#2:  An important announcement created 2018-09-17 17:46:43 by␣
↪slarse
[INFO] glassey-task-2/#2:   An important announcement created 2018-09-17 17:46:43 by␣
↪slarse
```

Other interesting arguments include `--all` for both open and closed issues, `--show-body` for showing the body of each issue, and `--author <username>` for filtering by author. There's not much more to it, see `repobee list-issues -h` for complete and up-to-date information on usage!

3.4 Cloning Repos in Bulk (the `clone` command)

It can at times be beneficial to be able to clone a bunch of student repos at the same time. It could for example be prudent to do this slightly after a deadline, as timestamps in a `git` commit can easily be altered (and are therefore not particularly trustworthy). Whatever your reason may be, it's very simple using the `clone` command. Again, assume that we have the `students.txt` file from *Set up student repositories*, and that we want to clone all student repos based on `task-1` and `task-2`.

```
$ repobee clone --mn task-1 task-2 --sf students.txt
[INFO] cloning into student repos ...
[INFO] Cloned into https://some-enterprise-host/repobee-demo/slarse-task-1
[INFO] Cloned into https://some-enterprise-host/repobee-demo/glassey-task-1
[INFO] Cloned into https://some-enterprise-host/repobee-demo/glassey-task-2
[INFO] Cloned into https://some-enterprise-host/repobee-demo/glennol-task-1
[INFO] Cloned into https://some-enterprise-host/repobee-demo/slarse-task-2
[INFO] Cloned into https://some-enterprise-host/repobee-demo/glennol-task-2
```

Splendid! That's really all there is to the basic functionality, the repos should now be in your current working directory. There is also a possibility to run automated tasks on cloned repos, such as running test suites or linters. If you're not satisfied with the tasks on offer, you can define your own. Read more about it in the *Plugins for RepoBee* section.

Note: For security reasons, RepoBee doesn't actually use `git clone` to clone repositories. Instead, RepoBee clones by initializing the repository and running `git pull`. The practical implication is that you can't simply enter a repository that's been cloned with RepoBee and run `git pull` to fetch updates. You will have to run `repobee clone` again in a different directory to fetch any updates students have made, alternatively simply delete to particular repositories you want to clone again and then run `repobee clone`.

3.5 Peer review (`assign-reviews`, `check-reviews` and `end-reviews` commands)

Peer reviewing is an important part of a programming curriculum, so of course RepoBee facilitates this! The relevant commands are `assign-reviews` and `end-reviews`. Like much of the other functionality in RepoBee, the peer review functionality is built around indirect access through teams with limited access privileges. In short, every student repo up for review gets an associated peer review team generated, which has `pull` access to the repo. Each student then gets added to `0 < N < num_students` peer review teams, and are to open a peer review issue in the associated repos. This is at least the the default. See *Selecting peer review allocation algorithm* for other available review allocation schemes.

3.5.1 Getting started with peer reviews using `assign-reviews`

The bulk of the work is performed by `assign-reviews`. Most of its arguments it has in common with the other commands of RepoBee. The only non-standard arguments are `--issue` and `--num-reviews`, the former of which we've actually already seen in the `open-issues` command (see *Opening Issues*). I will assume that both `--base-url` and `--org-name` are already configured in the configuration file (if you don't know what this mean, have a look at *Configuration file*). Thus, the only things we must specify are `--students`/`--students-file` and `--num-reviews` (`--issue` is optional, more on that later). Let's make a minimal call with the `assign-reviews` command, and then inspect the log output to figure out what happened. Recall that `students.txt` lists our three favorite students slarse, glassey and glennol (see *Set up student repositories*).

```
$ repobee assign-reviews --mn task-1 --sf students.txt --num-reviews 2
# step 1
[INFO] Created team slarse-task-1-review
[INFO] Created team glennol-task-1-review
[INFO] Created team glassey-task-1-review
# step 2
[INFO] Adding members glennol, glassey to team slarse-task-1-review
[INFO] Adding members glassey, slarse to team glennol-task-1-review
[INFO] Adding members slarse, glennol to team glassey-task-1-review
# steps 3 and 4, interleaved
[INFO] Opened issue glennol-task-1/#1-'Peer review'
[INFO] Adding team glennol-task-1-review to repo glennol-task-1 with 'pull' permission
[INFO] Opened issue glassey-task-1/#2-'Peer review'
[INFO] Adding team glassey-task-1-review to repo glassey-task-1 with 'pull' permission
[INFO] Opened issue slarse-task-1/#2-'Peer review'
[INFO] Adding team slarse-task-1-review to repo slarse-task-1 with 'pull' permission
```

The following steps were performed:

1. One review team per repo was created (`<student>-task-1-review`).
2. Two students were added to each review team. Note that these allocations are _random_. For obvious resons, there can be at most `num_students-1` peer reviews per repo. So, in this case, we are at the maximum.
3. An issue was opened in each repo with the title `Peer review`, and a body saying something like `You should peer review this repo.`. The review team students were assigned to the issue as well (although this is not apparent from the logging).
4. The review teams were added to their corresponding repoos with `pull` permission. This permission allows members of the team to view the repo and open issues, but they can't push to (and therefore can't modify) the repo.

That's it for the basic functionality. The intent is that students should open an issue in every repo they are to peer review, with a specific title. The issues can then be searched by title, and the `check-reviews` command can find which students have opened issues in the repositories they've been assigned to review. Now, let's talk a bit about that `--issue` argument.

Important: Assigning peer reviews gives the reviewers read-access to the repos they are to review. This means that if you use issues to communicate grades/feedback to your students, the reviewers will also see this feedback! It is therefore important to remove the peer review teams (see *Cleaning up with end-reviews*).

Specifying a custom issue

The default issue is really meant to be replaced with something more specific to the course and assignment. For example, say that there were five tasks in the `task-2` repo, and the students should review tasks 2 and 3 based on some criteria. It would then be beneficial to specify this in the peer review issue, so we'll write up our own little issue to replace the default one. Remember that the first line is taken to be the title, in exactly the same way as issue files are treated in *Opening Issues*.

```
Review of task-2

Hello! The students assigned to this issue have been tasked to review this
repo. Each of you should open _one_ issue with the title `Peer review` and
the following content:

```

(continues on next page)

(continued from previous page)

```
## Task 2
### Code style
Comments on code style, such as readability and general formatting.

### Time complexity
Is the algorithm O(n)? If not, try to figure out what time complexity it is
and point out what could have been done better.

## Task 3
### Code style
Comments on code style, such as readabilty and general formatting.
```

Assuming the file was saved as `issue.md`, we can now run the command specifying the issue like this:

```
$ repobee assign-reviews --mn task-2 --sf students.txt --num-reviews 2 --issue issue.
↪md
[INFO] Created team slarse-task-2-review
[INFO] Created team glennol-task-2-review
[INFO] Created team glassey-task-2-review
[INFO] Adding members glennol, glassey to team slarse-task-2-review
[INFO] Adding members glassey, slarse to team glennol-task-2-review
[INFO] Adding members slarse, glennol to team glassey-task-2-review
[INFO] Adding team glassey-task-2-review to repo glassey-task-2 with 'pull' permission
[INFO] Opened issue glassey-task-2/#8-'Review of task-2'
[INFO] Adding team glennol-task-2-review to repo glennol-task-2 with 'pull' permission
[INFO] Opened issue glennol-task-2/#8-'Review of task-2'
[INFO] Adding team slarse-task-2-review to repo slarse-task-2 with 'pull' permission
[INFO] Opened issue slarse-task-2/#9-'Review of task-2'
```

As you can tell from the last few lines, the title is the one specified in the issue, and not the default title as it was before. And that's pretty much it for setting up the peer review repos.

3.5.2 Checking review progress with `check-reviews`

The `check-reviews` command provides a quick and easy way of checking which students have performed their reviews. You provide it with the same information that you do for `assign-reviews`, but additionally also provide a regex to match against issue titles. The command then finds all of the associated review teams, and checks which students have opened issues with matching titles in their alloted repositories. Of course, this says *nothing* about the content of those issues: it purely checks that the issues have been opened at all. `--num-reviews` is also required here, as it is used as an expected value for how many reviews each student *should* be assigned to review. It is a simple but fairly effective way of detecting if students have simply left their review teams. Here's an example call:

```
$ repobee check-reviews --mn task-2 --sf students.txt --num-reviews 2 --title-regex
↪'\APeer review\Z'
[INFO] Processing glassey-task-2-review
[INFO] Processing glennol-task-2-review
[INFO] Processing slarse-task-2-review
reviewer        num done        num remaining   repos remaining
glennol         0               2               glassey-task-2,slarse-task-2
slarse          2               0
glassey         0               2               glennol-task-2,slarse-task-2
```

The output is color-coded in the terminal, making it easier to parse. I find this higly useful when doing peer reviews in a classroom settings, as I can check which students are done without having to ask them out loud every five minutes.

The next command lets you clean up review teams and thereby revoke reviewers' read access once reviews are over and done with.

3.5.3 Cleaning up with `end-reviews`

The one downside of using teams for access privileges is that we bloat the organization with a ton of teams. Once the deadline has passed and all peer reviews are done, there is little reason to keep them (in my mind). It can also often be a good idea to revoke the reviewers' access to reviewed repos if you yourself plan to provide feedback on the issue tracker, so as not to let the reviewers see it. Therefore, the `end-reviews` command can be used to remove all peer review teams for a given set of student repos, both cleaning up the organization and revoking reviewers' read access. Let's say that we're completely done with the peer reviews of `task-1`, and want to remove the review teams. It's as simple as:

```
$ repobee end-reviews --mn task-1 --sf students.txt
[INFO] Deleted team glennol-task-1-review
[INFO] Deleted team glassey-task-1-review
[INFO] Deleted team slarse-task-1-review
```

Warning: `end-reviews` *deletes* review allocations created by `assign-reviews`. This is an irreversible action. You cannot run `check-reviews` after running `end-reviews` for any given set of student repos, and there is no functionality for retrieving deleted review allocations. Only use `end-reviews` when reviews are truly done, **and** you have collected what results you need. If being able to backup and restore review allocations is something you need, please open an issue with a feature request on the issue tracker.

And that's it, the review teams are gone. If you also want to close the related issues, you can simply use the `close-issues` command for that (see *Closing Issues*). `end-reviews` plays one more important role: if you mess something up when assigning the peer reviews. The next section details how you can deal with such a scenario.

3.5.4 Messing up and getting back on track

Let's say you messed something up with allocating the peer reviews. For example, if you left out a student, there is no easy way to rectify the allocations such that that student is included. Let's say we did just that, and forgot to include the student `cabbage` in the reviews for `task-2` back at *Getting started with peer reviews using assign-reviews*. We then do the following:

1. Check if any reviews have already been posted. This can easily be performed with `repobee list-issues --mn task-2 --sf students.txt -r '^Peer review$'` (assuming the naming conventions were followed!). Take appropriate action if you find any reviews already posted (appropriate being anything you see fit to alleviate the situation of affected students possibly being assigned new repos to review).
2. Purge the review teams with `repobee end-reviews --mn task-2 --sf students.txt`
3. Close all review issues with `repobee close-issues --mn task-2 --sf students.txt -r '^Review of task-2$'`
4. Create a new `issue.md` file apologetically explaining that you messed up:

```
Review of task-2 (for real this time!)

Sorry, I messed up with the allocations previously. Disregard the previous
allocations (repo access has been revoked anyway).
```

5. Assign peer reviews again, with the new issue, with `repobee assign-reviews --mn task-2 --sf students.txt --num-reviews 2 --issue issue.md`

And that's it! Disaster averted.

3.5.5 Selecting peer review allocation algorithm

The default allocation algorithm is as described in *Peer review (assign-reviews, check-reviews and end-reviews commands)*, and is suitable for when reviewers do not need to interact with the students whom they review. This is however not always the case, sometimes it is beneficial for reviewers to to interact with reviewees (is that a word?), especially if the peer review is done in the classroom. Because of this, RepoBee also provides a _pairwise_ allocation scheme, which allocates reviews such that if student `A` reviews student `B`, then student `B` reviews student `A` (except for an `A->B->C->A` kind of deal in one group if there are an odd amount of students). This implemented as a plugin, so to run with this scheme, you add `-p pairwise` in front of the command.

```
$ repobee -p pairwise assign-reviews --mn task-1 --sf students.txt
```

Note that the pairwise algorithm ignores the `--num-reviews` argument, and will issue a warning if this is set (to anything but 1, but you should just not specify it). For more details on plugins in RepoBee, *Plugins for RepoBee*.

3.6 Plugins for RepoBee

RepoBee defines a fairly simple but powerful plugin system that allows programmers to hook into certain execution points. To read more about the details of these hooks (and how to write your own plugins), see the repobee-plug docs.

3.6.1 List of Plugins

This is a list of curated plugins that are safe to use. If you find a plugin for RepoBee not on this list, please get in touch and it may be added.

Internal (ship with RepoBee)

Plugins marked with `*` belong to the default plugins and are loaded automatically. These plugins can be used for reference on how to implement plugins, you can find the source code for them in the ext directory of the GitHub repo.

- `github*`
  - Allows for interfacing with GitHub instances.
- `configwizard*`
  - Adds the `config-wizard` command for editing the configuration file.
- `gitlab`
  - Allows for interfacing with GitLab instances (see *RepoBee and GitLab*).
- `javac`
  - Hooks into the `clone` command and runs `javac` on all Java code in every cloned student repository.
- `pylint`
  - Hooks into the `clone` command and runs `pylint` on all Python code in every cloned repository.
- `pairwise`
  - Hooks into the `assign-reviews` command and allocates students into pairs that internally review each other.

- `query` (EXPERIMENTAL!)
  - Adds the `query` command to RepoBee allowing the user to query hook results files for information.

External plugins (install separately)

External plugins that need to be installed separately from RepoBee. These are external either because they are complicated, or because their use cases are too specific for the core application.

- junit4
  - Hooks into the `clone` command and runs JUnit4 test classes on students' Java code.
- csvgrades
  - Finds issues matching user-specified conditions and reports them as grades into a CSV file. Useful for teachers who (themselves or via TAs) provide grading feedback in issues and need a way to automatically compose the results.
- feedback
  - Looks for local issue files following the naming convention `<STUDENT_REPO_NAME>.md` and opens them as issues in the respective student repos.
- gofmt
  - Simple plugin that runs `gofmt` on student's Go source code and reports whether or not it thinks the student code needs formatting or not.

3.6.2 Using Existing Plugins

You can specify which plugins you want to use either by adding them to the configuration file, or by specifying them on the command line. Personally, I find it most convenient to specify plugins on the command line. To do this, use `-p|--plug` option *before* any other options. The reson the plugins must go before any other options is that some plugins alter the command line interface of RepoBee, and must therefore be parsed separately. As an example, you can activate the *builtins* `javac` and `pylint` like this:

```
$ repobee -p pylint -p javac clone --mn task-1 --sf students.txt
```

This will clone the repos, and the run the plugins on the repos. You can also specify the default plugins you would like to use in the configuration file by adding the `plugins` option under the `[DEFAULT]` section. Here is an example of using the *builtins* `javac` and `pylint`.

```
[DEFAULTS]
plugins = javac, pylint
```

Like with all other configuration values, they are only used if no command line options are specified. If you have defaults specified, but want to run without any plugins, you can use the `--no-plugins` argument, which disables plugins.

Important: The order plugins are specified in is significant and defines the execution order of the plugins. This is useful for plugins that rely on the results of other plugins. This system for deciding execution order may be overhauled in the future, if anyone comes up with a better idea.

Some plugins can be further configured in the configuration file by adding new headers. See the documentation of the specific plugins for details on that.

3.6.3 Built-in API plugins

RepoBee ships with two API plugins, one for GitHub (`_repobee.ext.github`) and one for GitLab (`_repobee.ext.gitlab`). The GitHub plugin is loaded by default. If you use GitLab, you must specify the `gitlab` plugin either on the command line or in the configuration file.

3.6.4 Built-in subcommand plugins

The `config-wizard` command is actually a plugin, which loads by default. It's mostly implemented as a plugin for demonstrational purposes, showing how to add a command to RepoBee. See `_repobee.ext.configwizard` for the source code.

3.6.5 Built-in plugins for `repobee assign-reviews`

RepoBee ships with two plugins for the `assign-reviews` command. The first of these is located in the `defaults` plugin, and just randomly allocates student to review each other. The second plugin is the `pairwise` plugin. This plugin will divide `N` students into `N/2` groups of 2 students (and possibly one with 3 students, if `N` is odd), and have them peer review the other person in the group. The intention is to let students sit together and be able to ask questions regarding the repo they are peer reviewing. To use this allocation algorithm, simply specify the plugin with `-p pairwise` to override the default algorithm. Note that this plugin ignores the `--num-reviews` argument.

3.6.6 Built-in Plugins for `repobee clone`

RepoBee currently ships with two built-in plugins: `javac` and `pylint`. The former attempts to compile all `.java` files in each cloned repo, while the latter runs pylint on every `.py` file in each cloned repo. These plugins are mostly meant to serve as demonstarations of how to implement simple plugins in the `repobee` package itself.

`pylint`

The `pylint` plugin is fairly simple: it finds all `.py` files in the repo, and runs `pylint` on them individually. For each file `somefile.py`, it stores the output in the file `somefile.py.lint` in the same directory. That's it, the `pylint` plugin has no other features, it just does its thing.

Important: pylint must be installed and accessible by the script for this plugin to work!

`javac`

The `javac` plugin runs the Java compiler program `javac` on all `.java` files in the repo. Note that it tries to compile *all* files at the same time.

CLI Option

`javac` adds a command line option `-i|--ignore` to `repobee clone`, which takes a space-separated list of files to ignore when compiling.

Configuration

`javac` also adds a configuration file option `ignore` taking a comma-separated list of files, which must be added under the `[javac]` section. Example:

```
[DEFAULTS]
plugins = javac

[javac]
ignore = Main.java, Canvas.java, Other.java
```

Important: The `javac` plugin requires `javac` to be installed and accessible from the command line. All `JDK` distributions come with `javac`, but you must also ensure that it is on the PATH variable.

3.6.7 External Plugins

It's also possible to use plugins that are not included with RepoBee. Following the conventions defined in the repobee-plug docs, all plugins uploaded to PyPi should be named `repobee-<plugin>`, where `<plugin>` is the name of the plugin and thereby the thing to add to the `plugins` option in the configuration file. Any options for the plugin itself should be located under a header named `[<plugin>]`. For example, if I want to use the repobee-junit4 plugin, I first install it:

```
python3 -m pip install repobee-junit4
```

and then use for example this configuration file to activate the plugin, and define some defaults:

```
[DEFAULTS]
plugins = junit4

[junit4]
hamcrest_path = /absolute/path/to/hamcrest-1.3.jar
junit_path = /absolute/path/to/junit-4.12.jar
```

Important: If the configuration file exists, it *must* contain the `[DEFAULTS]` header, even if you don't put anything in that section. This is to minimize the risk of subtle misconfiguration errors by novice users. If you only want to configure plugins, just add the `[DEFAULTS]` header by itself, without options, to meet this requirement.

3.7 Migrate repositories into the target (or master) organization (`migrate` command)

Migrating repositories into an organization can be useful in a few cases. You may have repos that should be accessible to students and need to be moved across course rounds, or you might be storing your master repos in the target organization and need to migrate them for each new course round. To migrate repos into the target organization, they must be local on disc. Assuming we have the repos `task-1` and `task-2` in the current working directory (i.e. local repos), all we have to do is this:

Note: Prior to v1.4.0, the `migrate` command also accepted urls with the `-mu` option. This functionality was abruptly removed due to implementation issues, and is unlikely to appear again because of its limited use.

```
$ repobee migrate --mn task-1 task-2
[INFO] cloning into file:///some/directory/path/task-1
[INFO] cloning into file:///some/directory/path/task-2
[INFO] created repobee-demo/task-1
[INFO] created repobee-demo/task-2
[INFO] pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/task-1 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/task-2 master
[INFO] done!
```

Important: If you want to use this command to migrate repos into a master organization, you must specify it with the `--org-name` option here (instead of the `--master-org-name`).

What happens here is pretty straightforward, except for the local repos being cloned, which is an implementation detail that does not need to be thought further of. Note that only the defualt branch is actually migrated, and pushed to `master` in the new repo. Local repos are pushed to the `master` branch of the remote repo. Migrating several branches is something that we've never had a need to do, but if you do, please open an issue on GitHub with a feature request. `migrate` is perfectly safe to run several times, in case you think you missed something, or need to update repos. Running the same thing again without changing the local repos yields the following output:

```
$ repobee migrate --mn task-1 task-2
[INFO] cloning into file:///some/directory/path/task-1
[INFO] cloning into file:///some/directory/path/task-2
[INFO] repobee-demo/task-1 already exists
[INFO] repobee-demo/task-2 already exists
[INFO] pushing, attempt 1/3
[INFO] https://some-enterprise-host/repobee-demo/task-1 master is up-to-date
[INFO] https://some-enterprise-host/repobee-demo/task-2 master is up-to-date
[INFO] done!
```

In fact, all RepoBee commands that deal with pushing to or cloning from repos in some way are safe to run over and over. This is mostly because of how Git works, and has little to do with RepoBee itself.

3.8 Group assignments

Important: The peer review commands (see *Peer review (assign-reviews, check-reviews and end-reviews commands)*) do not currently support group assignments.

RepoBee supports group assignments such that multiple students are assigned to the same student repositories. To put students in a group, they need to be entered on the same line in the students file, separated by spaces. This is the only way to group students, the `-s` option on the command line does not support groups. As an example, if `glassey` and `slarse` should be in one group, and `glennol` solo, the following students file would work:

```
glassey slarse
glennol
```

There is no difference in using RepoBee with student groups in the student file. For example, running the setup command from *Set up student repositories* would then have the following result:

```
$ repobee setup --mn task-1 task-2 --sf students.txt
[INFO] cloning into master repos ...
[INFO] cloning into file:///home/slarse/tmp/task-1
[INFO] cloning into file:///home/slarse/tmp/task-2
[INFO] created team glennol
[INFO] created team glassey-slarse
[INFO] adding members glennol to team glennol
[WARNING] user glennol does not exist
[INFO] adding members glassey, slarse to team glassey-slarse
[INFO] creating student repos ...
[INFO] created repobee-demo/glennol-task-1
[INFO] created repobee-demo/glassey-slarse-task-1
[INFO] created repobee-demo/glennol-task-2
[INFO] created repobee-demo/glassey-slarse-task-2
[INFO] pushing files to student repos ...
[INFO] pushing, attempt 1/3
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glassey-slarse-task-
↪2 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glassey-slarse-task-
↪1 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glennol-task-2 master
[INFO] Pushed files to https://some-enterprise-host/repobee-demo/glennol-task-1 master
```

Note the naming convention for group repos: `<student-1>-<student-2>-[...]-<master-repo-name>`. The associated teams follow the same convention, but without the trailing `-<master-repo-name>`. And that is all you need to know to start doing group assignments!

Warning: The naming scheme has a weakness: it can create fairly long names, and GitHub has a hard limit for repo names at 100 characters. RepoBee will therefore crash (on purpose) if a Team or repo name exceeds 100 characters. There is no workaround for this problem at the moment.

3.9 RepoBee and GitLab

As of v2.3.0, RepoBee fully supports GitLab for all commands, both on https://gitlab.com and on self-hosted GitLab instances. The functionality is new, so please report any bugs you find on the *issue tracker <https://github.com/repobee/repobee/issues/new>*. All of RepoBee's system tests run against a GitLab instances, so I have every intention of continuing to improve the GitLab experience.

Note: GitLab support is currently in alpha, and may not yet be sufficiently stable for production use. Please report any issues on the issue tracker

Important: RepoBee requires GitLab 11.11 or later. This is only relevant if you have a self-hosted GitLab instance.

3.9.1 GitLab terminology

RepoBee uses GitHub terminology, as GitHub is the primary platform. It is however simple to map the terminology between the two platforms as follows:

GitHub	GitLab
Organization	Group
Team	Subgroup
Repository	Project
Issue	Issue

So, if you read "target organization" in the documentation, that translates directly to "target group" when using GitLab. Although there are a few practical differences, the concepts on both platforms are similar enough that it makes no difference as far as using RepoBee goes. You can read more about differences and similarities in this GitLab blog post.

3.9.2 How to use RepoBee with GitLab

You must use the `gitlab` plugin for RepoBee to be able to interface with GitLab. See *Using Existing Plugins* for instructions on how to use plugins. Provide the url to a GitLab instance host (*not* to the api endpoint, just to the host) as an argument to `--bu|--base-url`, or put it in the config file as the value for option `base_url`. Other than that, there are a few important differences between GitHub and GitLab that the user should be aware of.

- As noted, the base url should be provided to the host of the GitLab instance, and not to any specific endpoint (as is the case when using GitHub). When using `github.com` for example, the url should be provided as `base_url = https://gitlab.com` in the config.
- The `org-name` and `master-org-name` arguments should be given the *path* of the respective groups. If you create a group with a long name, GitLab may shorten the path automatically. For example, I created the group `repobee-master-repos`, and it got the path `repobee-master`. You can find your path by going to the landing page of your group and checking the URL: the path is the last part. You can change the path manually by going to your group, then *Settings->General->Path,transfer,remove* and changing the group path.

Getting an access token for GitLab

Creating a personal access token token for a GitLab API is just as easy as creating one for GitHub. Just follow these instructions. The scopes you need to tick are `api`, `read_user`, `read_repository` and `write_repository`. That's it!

3.9.3 Roadmap

The roadmap for GitLab support is listed below. As GitLab is now fully supported, this serves only as a record of history (and to not break links I may have put elsewhere and then forgotten about :)).

Command	Status	ETA/Added in
show-config	Done	N/A (not platform dependent)
setup	Done	v1.5.0
update	Done	v1.5.0
clone	Done	v1.5.0
migrate	Done	v1.6.0
open-issues	Done	v1.6.0
close-issues	Done	v1.6.0
list-issues	Done	v1.6.0
assign-reviews	Done	v2.3.0
end-reviews	Done	v2.3.0
check-reviews	Done	v2.3.0
verify-settings	Done	v2.3.0

CHAPTER 4

Configuration

RepoBee does not *have* to be configured as all arguments can be provided on the command line, but doing so becomes very tedious, very quickly. It's typically a good idea to at least configure the *Access token*, as well as the GitHub base url (for the API) and your GitHub username (see *Configuration file*).

Important: The *RepoBee User Guide* expects there to be a configuration file as described in *Getting started (the show-config, verify-settings and setup commands)*.

4.1 Access token

For repobee to work at all, it needs a *Personal Access Token*. See the GitHub access token docs for how to create a token. Make sure that it has the `repo` and `admin:org` scopes. There are two ways to hand the token to repobee:

1. Put it in the `REPOBEE_TOKEN` environment variable. - On a unix system, this is as simple as `export REPOBEE_TOKEN=<YOUR_TOKEN>`
2. Put it in the configuration file (see *Configuration file*).

4.2 Configuration file

An optional configuration file can be added, specifying defaults for several of the most frequently used cli options line options. This is especially useful for teachers ant TAs who are managing repos for a single course (and, as a consequence, a single organization).

```
[DEFAULTS]
base_url = https://some-api-v3-url
user = YOUR_USERNAME
org_name = ORGANIZATION_NAME
master_org_name = MASTER_ORGANIZATION_NAME
```

(continues on next page)

(continued from previous page)

```
students_file = STUDENTS_FILE_ABSOLUTE_PATH
token = SUPER_SECRET_TOKEN
```

Important: If the configuration file exists, it *must* contain the `[DEFAULTS]` header. This is to minimize the risk of misconfiguration by novice users.

To find out where to place the configuration file (and what to name it), run `repobee show-config`. The configuration file can also be used to configure `repobee` plugins. See the *Using Existing Plugins* section for more details.

Important: Do note that the configuration file contains only default values. Specifying any of the parameters on the command line will override the configuration file's values.

Note: You can run `repobee verify-settings` to verify the basic configuration. This will check the most important settings configurable in `DEFAULTS`.

CHAPTER 5

CLI documentation

CHAPTER 6

Contributing to RepoBee

This article contains technical information on how to contribute to RepoBee. If you haven't alredy, first read the information in the CONTRIBUTING.md in the repo, which details how to submit a proposal. If you've done that, this article will tell you more about technical details.

6.1 Setting up a Development Environment

6.1.1 Basic Environment to Run Unit Tests

The most rudimentary development environment is easy to set up. There are three tasks to accomplish:

- *Fork the repository <https://help.github.com/en/articles/fork-a-repo>* and clone your fork.
- Setup a Python virtual environment and install the project with test dependencies.
- Install the pre-commit hooks

So, first *fork the repository <https://help.github.com/en/articles/fork-a-repo>* and clone your fork down to disk.

```
# substitute USER for your username
$ git clone git@github.com:USER/repobee.git
```

Then, you need to set up a virtual environment in the newly cloned repository. I'm using `pipenv` here, but you can use something else if you have other preferences.

```
# install pipenv for the local user
$ python3 -m pip install --user pipenv
# move into the repobee directory and install the repobee package with pipenv
$ cd repobee
$ python3 -m pipenv install -e .[TEST]
```

The last thing takes a while, so just be patient. When it's done, you can verify that everything was installed correctly by running the tests in the virtual environment.

```
$ python3 -m pipenv run pytest tests/unit_tests
```

Everything should pass. Now, you can run any command in the virtualenv by prepending it with `python3 -m pipenv run`. However, it is often more convenient to "enter" the virtual environment with `python3 -m pipenv shell`, and type `exit` to exit it. Then, you can just type in your Python commands as usual, and the virtual environment's Python program will be used.

Pre-commit Hooks

Finally, you should also install the pre-commit hooks that come with RepoBee. They make some rudimentary checks to primarily code style before a commit can be recorded. They require the `pre-commit` package. I recommend installing this *outside* of the virtual environment so that hooks can run even if you are not in the virtual environment shell. In the root of the project, run:

And that's it, the environment is all set up!

6.1.2 Full Environment to Run Integration/System Tests

To also run the integration/system tests located in `tests/integration_tests`, you need to have Docker and Docker Compose installed, and the Docker daemon (service) must be running. Installing these utilities will vary by distribution, here are a few examples:

```
# Arch Linux
$ sudo pacman -Sy docker docker-compose
# Ubuntu
$ sudo apt install docker docker-compose
# CentOS/REHL
$ sudo yum -y install epel-release # docker-compose is in the EPEL repos
$ sudo yum -y install docker docker-compose
```

Activating the Docker daemon also differs by distribution, but if you have `systemd`, it looks like this:

```
sudo systemctl start docker   # start ASAP
sudo systemctl enable docker  # start automatically on startup
```

Then, enter the `tests/integration_tests` directory and run the `startup.sh` script (you must be run **in** that directory and run the scrip, it's not a very robust script :D).

```
$ cd tests/integration_tests
$ ./startup.sh
```

This may take a long time to complete the first time, but there should always be output indicating that something is happening. This whole thing starts a local GitLab instance to run tests against.

Important: The GitLab instance may start automatically on startup after running the `startup.sh` script. To turn it off permanently, run `docker-compose down` in the `tests/integration_tests` directory.

Now the infrastructure needed for the integration tests is there. To actually run the integration tests, you first need to build the test container. In the root of the project, run:

```
$ sudo docker build -t repobee:test -f Dockerfile.test .
```

Important: Every time you change something in the production code, the test container must be rebuilt!

Then it's just a matter of running the integration tests (also from the root of the project).

```
$ sudo REPOBEE_NO_VERIFY_SSL='true' pytest tests/integration_tests/integration_tests.
↪py
```

This usually takes 10-20 minutes, depending on your hardware. To run just a subset of the tests, specify the `-k` option at the end, and follow with the name of a test class or a specific test. For example, to *only* run the TestUpdate class, you add `-k TestUpdate` to the end of the above command.

Note: If your user is part of the docker group, you do not need `sudo` for the `docker` and `docker-compose` commands.

6.2 Code Style

RepoBee follows a fairly strict code style, which is *mostly* enforced by the *Pre-commit Hooks*. So make sure you install them. The code is formatted by *Black <https://github.com/psf/black>*, and you have no say in that: Black does it the way it wants. What Black does not handle is docstrings. Any public function must have a docstring, complete with type annotations and argument+return value descriptions. Here are two examples:

Listing 1: Docstring examples

```
def func_without_return_value(int_param: int, string_param: str) -> None:
    """What the function does.

    Args:
        int_param: Description of the int_param.
        string_param: Description of the string_param.
    """

def func_with_return_value(int_param: int, string_param: str) -> str:
    """What the function does.

    Args:
        int_param: Description of the int_param.
        string_param: Description of the string_param.
    Returns:
        Description of return value.
    """
```

6.3 Contributing to Docs

To be able to build the documentation, you must install the dependencies liste in `requirements/docs.txt`, in addition to installing the package itself. In your virtual environment, run the following from the root of the repository:

```
$ pip install -r requirements/docs.txt
```

Then, to build the documentation, enter the `docs` directory and run `make html`.

```
$ cd docs
$ make html
```

This will produce the documentation in `docs/_build/html`, with the landing page being `docs/_build/html/index.html`.

CHAPTER 7

RepoBee Module Reference

7.1 command

7.2 cli

7.3 config

7.4 exception

7.5 git

7.6 tuples

7.7 util

7.8 Core plugins

7.8.1 defaults

7.8.2 github

7.8.3 gitlab

7.8.4 pairwise

7.9 Extension plugins

7.9.1 javac

7.9.2 pylint

CHAPTER 8

Indices and tables

- genindex
- modindex
- search