{cookiecutter.app_name} Documentation

Release 0.10.0

{cookiecutter.author_name}

Dec 15, 2019

Contents:

CHAPTER 1

Plugin system overview

1.1 Conventions

For RepoBee to discover a plugin and its hooks, the following conventions need to be adhered to:

1. The PyPi package should be named `repobee-<plugin>`, where `<plugin>` is the name of the plugin.
2. The actual Python package (i.e. the directory in which the source files are located) should be called `repobee_<plugin>`. In other words, replace the hyphen in the PyPi package name with an underscore.
3. The Python module that defines the plugin's hooks/hook classes should be name `<plugin>.py`.

For an example plugin that follows these conventions, have a look at repobee-junit4. Granted that the plugin follows these conventions and is installed, it can be loaded like any other RepoBee plugin (see Using Existing Plugins).

1.2 Hooks

There are two types of hooks in RepoBee: *core hooks* and *extension hooks*.

1.2.1 Core hooks

Core hooks provide core functionality for RepoBee, and always have a default implementation in `repobee.ext.defaults`. Providing a different plugin implementation will override this behavior, thereby changing some core part of RepoBee. In general, only one implementation of a core hook will run per invocation of RepoBee. All core hooks are defined in *`repobee_plug.corehooks`*.

Important: Note that the default implementations in `repobee.ext.defaults` may simply be *imported* into the module. They are not necessarily defined there.

1.2.2 Extension hooks

Extension hooks extend the functionality of RepoBee in various ways. Unlike the core hooks, there are no default implementations of the extension hooks, and multiple implementations can be run on each invocation of RepoBee. All extension hooks are defined in *repobee_plug.exthooks*.

CHAPTER 2

Implementing hooks and writing internal plugins

Implementing a hook is fairly simple, and works the same way regardless of what type of hook it is (core or extension). If you are working with your own fork of RepoBee, all you have to do is write a small module implementing some hooks, and drop it into the `repobee.ext` sub-package (i.e. the in directory `repobee/ext` in the RepoBee repo).

There are two ways to implement hooks: as standalone functions or wrapped in a class. In the following two sections, we'll implement the *`act_on_cloned_repo()`* extension hook using both techniques. Let's call the plugin `exampleplug` and make sure it adheres to the plugin conventions.

2.1 Hook functions in a plugin class

Wrapping hook implementations in a class inheriting from *`Plugin`* is the recommended way to write plugins for RepoBee. The class does some checks to make sure that all public functions have hook function names, which comes in handy if you are in the habit of misspelling stuff (aren't we all?). Doing it this way, `exampleplug.py` would look like this:

Listing 1: exampleplug.py

```python
import pathlib
import os
from typing import Union

import repobee_plug as plug

PLUGIN_NAME = 'exampleplug'

class ExamplePlugin(plug.Plugin):
    """Example plugin that implements the act_on_cloned_repo hook."""

    def act_on_cloned_repo(
        self, path: Union[str, pathlib.Path], api,
    ) -> plug.HookResult:
        """Do something with a cloned repo.
```

(continues on next page)

(continued from previous page)

```
        Args:
            path: Path to the student repo.
            api: A platform API instance.
        Returns:
            a plug.HookResult specifying the outcome.
        """
        return plug.HookResult(
            hook=PLUGIN_NAME, status=plug.Status.WARNING, msg="This isn't quite done")
```

Dropping `exampleplug.py` into the `repobee.ext` package and running `repobee -p exampleplug clone [ADDITIONAL ARGS]` should give some not-so-interesting output from the plugin.

The name of the class really doesn't matter, it just needs to inherit from *Plugin*. The name of the module and hook functions matter, though. The name of the module must be the plugin name, and the hook functions must have the precise names of the hooks they implement. In fact, all public methods in a class deriving from *Plugin* must have names of hook functions, or the class will fail to be created. You can see that the hook returns a `HookResult`. This is used for reporting the results in RepoBee, and is entirely optional (not all hooks support it, though). Do note that if `None` is returned instead, RepoBee will not report anything for the hook. It is recommended that hooks that can return `HookResult` do. For a comprehensive example of an internal plugin implemented with a class, see the built-in javac plugin.

2.2 Standalone hook functions

Using standalone hook functions is recommended only if you don't want the safety net provided by the *Plugin* metaclass. It is fairly straightforward: simply mark a function with the `repobee_plug.repobee_hook` decorator. With this approach, `exampleplug.py` would look like this:

Listing 2: exampleplug.py

```
import pathlib
import os
from typing import Union

import repobee_plug as plug

PLUGIN_NAME = 'exampleplug'

@plug.repobee_hook
def act_on_cloned_repo(path: Union[str, pathlib.Path]) -> plug.HookResult:
    """Do something with a cloned repo.

    Args:
        path: Path to the student repo.
    Returns:
        a plug.HookResult specifying the outcome.
    """
    return plug.HookResult(
        hook=PLUGIN_NAME, status=plug.Status.WARNING, msg="This isn't quite done")
```

Again, dropping `exampleplug.py` into the `repobee.ext` package and running `repobee -p exampleplug clone [ADDITIONAL ARGS]` should give some not-so-interesting output from the plugin. For a more practical example of a plugin implemented using only a hook function, see the built-in pylint plugin.

CHAPTER 3

Writing external plugins (recommended and easy!)

Writing an external plugin is really easy using the repobee-plugin-cookiecutter template. First of all, you need to install cookiecutter. It's on PyPi and installs just the same as `repobee` with `pip install cookiecutter` (with whatever flags you like to use). Now, running `python3 -m cookiecutter gh:repobee/repobee-plugin-cookiecutter` will give you some prompts to answer. If you want to create a plugin called `exampleplug`, it looks something like this:

```
$ python3 -m cookiecutter gh:repobee/repobee-plugin-cookiecutter
author []: Your Name
email []: email@address.com
github_username []: your_github_username
plugin_name []: exampleplug
short_description []: An example plugin!
```

This will result in a directory called `repobee-exampleplug`, containing a fully functioning (albeit quite useless) external plugin. If you do `cd exampleplug` and then run `pip install -e .`, you will install the plugin locally. You can then use it like any of the built-in plugins, as described in Using Existing Plugins. To actually implement the behavior that you want, edit the file `repobee-exampleplug/repobee_exampleplug/exampleplug.py` to implement the hooks you want.

CHAPTER 4

`repobee_plug` Module Reference

CHAPTER 5

Public API

The public API of `repobee_plug` is what's intended to be used directly in plugins.

class `repobee_plug.`**`Plugin`**

Base class for plugin classes. For plugin classes to be picked up by `repobee`, they must inherit from this class.

Public methods must be hook methods, i.e. implement the specification of one of the hooks defined in *`PeerReviewHook`* or *`CloneHook`*. If there are any other public methods, an error is raised on class creation. As long as the method has the correct name, it will be recognized as a hook method.

The signature of the method is not checked until the hook is registered by the `repobee_plug.manager` (an instance of `pluggy.manager.PluginManager`). Therefore, when testing a plugin, it is a good idea to include a test where it is registered with the manager to ensure that it has the correct signatures.

Private methods (i.e. methods prefixed with _) carry no such restrictions.

class `repobee_plug.`**`HookResult`**

Container for storing results from hooks.

class `repobee_plug.`**`Status`**

Status codes enum.

class `repobee_plug.`**`ExtensionParser`**

An ArgumentParser specialized for RepoBee extension commands.

class `repobee_plug.`**`ExtensionCommand`**

Class defining an extension command for the RepoBee CLI.

class `repobee_plug.`**`ReviewAllocation`**(*review_team*, *reviewed_team*)

`review_team`

Alias for field number 0

`reviewed_team`

Alias for field number 1

class `repobee_plug.`**`Review`**(*repo*, *done*)

done
Alias for field number 1

repo
Alias for field number 0

class `repobee_plug.`**Team**
Wrapper class for a Team API object.

class `repobee_plug.`**TeamPermission**
Enum specifying team permissions on creating teams. On GitHub, for example, this can be e.g. *push* or *pull*.

class `repobee_plug.`**Issue**
Wrapper class for an Issue API object.

static from_dict(*asdict*)
Take a dictionary produced by Issue.to_dict and reconstruct the corresponding instance. The `implementation` field is lost in a to_dict -> from_dict roundtrip.

Return type *`Issue`*

to_dict()
Return a dictionary representation of this namedtuple, without the `implementation` field.

class `repobee_plug.`**Repo**
Wrapper class for a Repo API object.

class `repobee_plug.`**Issue**
Wrapper class for an Issue API object.

static from_dict(*asdict*)
Take a dictionary produced by Issue.to_dict and reconstruct the corresponding instance. The `implementation` field is lost in a to_dict -> from_dict roundtrip.

Return type *`Issue`*

to_dict()
Return a dictionary representation of this namedtuple, without the `implementation` field.

class `repobee_plug.`**IssueState**
Enum specifying a possible issue state.

class `repobee_plug.`**API**(*base_url*, *token*, *org_name*, *user*)
API base class that all API implementations should inherit from. This class functions similarly to an abstract base class, but with a few key distinctions that affect the inheriting class.

1. Public methods *must* override one of the public methods of `APISpec`. If an inheriting class defines any other public method, an `APIError` is raised when the class is defined.
2. All public methods in `APISpec` have a default implementation that simply raise a `NotImplementedError`. There is no requirement to implement any of them.

exception `repobee_plug.`**ExtensionCommandError**
Raise when an :py:class:~repobee_plug.containers.ExtensionCommand: is incorrectly defined.

exception `repobee_plug.`**HookNameError**
Raise when a public method in a class that inherits from *`Plugin`* does not have a hook name.

exception `repobee_plug.`**PlugError**
Base class for all repobee_plug exceptions.

`repobee_plug.`**json_to_result_mapping**(*json_string*)
Deserialize a JSON string to a mapping `repo_name: str -> hook_results: List[HookResult]`

Return type `Mapping[str, List[HookResult]]`

`repobee_plug.`**`result_mapping_to_json`**(*result_mapping*)

Serialize a result mapping `repo_name: str -> hook_results: List[HookResult]` to JSON.

Return type `str`

`class` `repobee_plug.`**`BaseParser`**

Enumeration of base parsers that an extension command can request.

`repobee_plug.`**`generate_repo_name`**(*team_name*, *master_repo_name*)

Construct a repo name for a team.

Parameters

- **`team_name`** (`str`) – Name of the associated team.
- **`master_repo_name`** (`str`) – Name of the template repository.

Return type `str`

`repobee_plug.`**`generate_repo_names`**(*team_names*, *master_repo_names*)

Construct all combinations of generate_repo_name(team_name, master_repo_name) for the provided team names and master repo names.

Parameters

- **`team_names`** (`Iterable[str]`) – One or more names of teams.
- **`master_repo_names`** (`Iterable[str]`) – One or more names of master repositories.

Return type `Iterable[str]`

Returns a list of repo names for all combinations of team and master repo.

`repobee_plug.`**`generate_review_team_name`**(*student*, *master_repo_name*)

Generate a review team name.

Parameters

- **`student`** (`str`) – A student username.
- **`master_repo_name`** (`str`) – Name of a master repository.

Return type `str`

Returns a review team name for the student repo associated with this master repo and student.

CHAPTER 6

Internal API

The internal API of `repobee_plug` should only be used internally.

6.1 apimeta

Metaclass for API implementations.

`APIMeta` defines the behavior required of platform API implementations, based on the methods in *`APISpec`*. With platform API, we mean for example the GitHub REST API, and the GitLab REST API. The point is to introduce another layer of indirection such that higher levels of RepoBee can use different platforms in a platform-independent way. *`API`* is a convenience class so consumers don't have to use the metaclass directly.

Any class implementing a platform API should derive from *`API`*. It will enforce that all public methods are one of the method defined py *`APISpec`*, and give a default implementation (that just raises NotImplementedError) for any unimplemented API methods.

class `repobee_plug.apimeta.`**`API`**(*base_url*, *token*, *org_name*, *user*)

API base class that all API implementations should inherit from. This class functions similarly to an abstract base class, but with a few key distinctions that affect the inheriting class.

1. Public methods *must* override one of the public methods of *`APISpec`*. If an inheriting class defines any other public method, an `APIError` is raised when the class is defined.
2. All public methods in *`APISpec`* have a default implementation that simply raise a `NotImplementedError`. There is no requirement to implement any of them.

class `repobee_plug.apimeta.`**`APIMeta`**

Metaclass for an API implementation. All public methods must be a specified api method, but all api methods do not need to be implemented.

class `repobee_plug.apimeta.`**`APIObject`**

Base wrapper class for platform API objects.

class `repobee_plug.apimeta.`**`APISpec`**(*base_url*, *token*, *org_name*, *user*)

Wrapper class for API method stubs.

Important: This class should not be inherited from directly, it serves only to document the behavior of a platform API. Classes that implement this behavior should inherit from *API*.

add_repos_to_review_teams (*team_to_repos*, *issue=None*)
Add repos to review teams. For each repo, an issue is opened, and every user in the review team is assigned to it. If no issue is specified, sensible defaults for title and body are used.

Parameters

- **team_to_repos** (Mapping[str, Iterable[str]]) – A mapping from a team name to an iterable of repo names.
- **issue** (Optional[*Issue*]) – An optional Issue tuple to override the default issue.

Return type None

close_issue (*title_regex*, *repo_names*)
Close any issues in the given repos in the target organization, whose titles match the title_regex.

Parameters

- **title_regex** (str) – A regex to match against issue titles.
- **repo_names** (Iterable[str]) – Names of repositories to close issues in.

Return type None

create_repos (*repos*)
Create repos in the target organization according the those specced by the `repos` argument. Repos that already exist are skipped.

Parameters **repos** (Iterable[*Repo*]) – Repos to be created.

Return type List[str]

Returns A list of urls to the repos specified by the `repos` argument, both those that were created and those that already existed.

delete_teams (*team_names*)
Delete all teams in the target organizatoin that exactly match one of the provided `team_names`. Skip any team name for which no match is found.

Parameters **team_names** (Iterable[str]) – A list of team names for teams to be deleted.

Return type None

ensure_teams_and_members (*teams*, *permission=<TeamPermission.PUSH: 'push'>*)
Ensure that the teams exist, and that their members are added to the teams.

Teams that do not exist are created, teams that already exist are fetched. Members that are not in their teams are added, members that do not exist or are already in their teams are skipped.

Parameters

- **teams** (Iterable[*Team*]) – A list of teams specifying student groups.
- **permission** (*TeamPermission*) – The permission these teams (or members of them) should be given in regards to associated repositories.

Return type List[*Team*]

Returns A list of Team API objects of the teams provided to the function, both those that were created and those that already existed.

`extract_repo_name`(*repo_url*)

Extract a repo name from the provided url.

Parameters **`repo_url`** (`str`) – A URL to a repository.

Return type `str`

Returns The name of the repository corresponding to the url.

`get_issues`(*repo_names*, *state=<IssueState.OPEN: 'open'>*, *title_regex=''*)

Get all issues for the repos in repo_names an return a generator that yields (repo_name, issue generator) tuples. Will by default only get open issues.

Parameters

- **`repo_names`** (`Iterable[str]`) – An iterable of repo names.
- **`state`** (*`IssueState`*) – Specifies the state of the issue.
- **`title_regex`** (`str`) – If specified, only issues matching this regex are
- **`Defaults to the empty string`** (*`returned.`*) –

Return type `Generator[Tuple[str, Generator[Issue, None, None]], None, None]`

Returns A generator that yields (repo_name, issue_generator) tuples.

`get_repo_urls`(*master_repo_names*, *org_name=None*, *teams=None*)

Get repo urls for all specified repo names in the organization. As checking if every single repo actually exists takes a long time with a typical REST API, this function does not in general guarantee that the urls returned actually correspond to existing repos.

If the `org_name` argument is supplied, urls are computed relative to that organization. If it is not supplied, the target organization is used.

If the *teams* argument is supplied, student repo urls are computed instead of master repo urls.

Parameters

- **`master_repo_names`** (`Iterable[str]`) – A list of master repository names.
- **`org_name`** (`Optional[str]`) – Organization in which repos are expected. Defaults to the target organization of the API instance.
- **`teams`** (`Optional[List[Team]]`) – A list of teams specifying student groups. Defaults to None.

Return type `List[str]`

Returns a list of urls corresponding to the repo names.

`get_review_progress`(*review_team_names*, *teams*, *title_regex*)

Get the peer review progress for the specified review teams and student teams by checking which review team members have opened issues in their assigned repos. Only issues matching the title regex will be considered peer review issues. If a reviewer has opened an issue in the assigned repo with a title matching the regex, the review will be considered done.

Note that reviews only count if the student is in the review team for that repo. Review teams must only have one associated repo, or the repo is skipped.

Parameters

- **`review_team_names`** (`Iterable[str]`) – Names of review teams.
- **`teams`** (`Iterable[Team]`) – Team API objects specifying student groups.

- **title_regex** (str) – If an issue title matches this regex, the issue is considered a potential peer review issue.

Return type Mapping[str, List]

Returns a mapping (reviewer -> assigned_repos), where reviewer is a str and assigned_repos is a `_repobee.tuples.Review`.

get_teams()
Get all teams related to the target organization.

Return type List[*Team*]

Returns A list of Team API object.

open_issue(*title*, *body*, *repo_names*)
Open the specified issue in all repos with the given names, in the target organization.

Parameters

- **title** (str) – Title of the issue.
- **body** (str) – Body of the issue.
- **repo_names** (Iterable[str]) – Names of repos to open the issue in.

Return type None

static verify_settings(*user*, *org_name*, *base_url*, *token*, *master_org_name=None*)
Verify the following (to the extent that is possible and makes sense for the specifi platform):

1. Base url is correct
2. The token has sufficient access privileges
3. **Target organization (specifiend by `org_name`) exists**
    - If master_org_name is supplied, this is also checked to exist.
4. **User is owner in organization (verify by getting**
    - If master_org_name is supplied, user is also checked to be an owner of it.

organization member list and checking roles)

Should raise an appropriate subclass of `_repobee.exception.APIError` when a problem is encountered.

Parameters

- **user** (str) – The username to try to fetch.
- **org_name** (str) – Name of the target organization.
- **base_url** (str) – A base url to a github API.
- **token** (str) – A secure OAUTH2 token.
- **org_name** – Name of the master organization.

Returns True if the connection is well formed.

Raises `_repobee.exception.APIError`

class repobee_plug.apimeta.**Issue**
Wrapper class for an Issue API object.

static from_dict(*asdict*)
Take a dictionary produced by Issue.to_dict and reconstruct the corresponding instance. The `implementation` field is lost in a to_dict -> from_dict roundtrip.

Return type *Issue*

to_dict()
Return a dictionary representation of this namedtuple, without the `implementation` field.

class `repobee_plug.apimeta.`**IssueState**
Enum specifying a possible issue state.

class `repobee_plug.apimeta.`**Repo**
Wrapper class for a Repo API object.

class `repobee_plug.apimeta.`**Team**
Wrapper class for a Team API object.

class `repobee_plug.apimeta.`**TeamPermission**
Enum specifying team permissions on creating teams. On GitHub, for example, this can be e.g. *push* or *pull*.

`repobee_plug.apimeta.`**check_init_params**(*reference_params*, *compare_params*)
Check that the compare __init__'s parameters are a subset of the reference class's version.

`repobee_plug.apimeta.`**check_parameters**(*reference*, *compare*)
Check if the parameters match, one by one. Stop at the first diff and raise an exception for that parameter.

An exception is made for __init__, for which the compare may be a subset of the reference in no particular order.

`repobee_plug.apimeta.`**methods**(*attrdict*)
Return all public methods and __init__ for some class.

`repobee_plug.apimeta.`**parameters**(*function*)
Extract parameter names and default arguments from a function.

6.2 pluginmeta

class `repobee_plug.pluginmeta.`**Plugin**
Base class for plugin classes. For plugin classes to be picked up by `repobee`, they must inherit from this class.

Public methods must be hook methods, i.e. implement the specification of one of the hooks defined in *PeerReviewHook* or *CloneHook*. If there are any other public methods, an error is raised on class creation. As long as the method has the correct name, it will be recognized as a hook method.

The signature of the method is not checked until the hook is registered by the `repobee_plug.manager` (an instance of `pluggy.manager.PluginManager`). Therefore, when testing a plugin, it is a good idea to include a test where it is registered with the manager to ensure that it has the correct signatures.

Private methods (i.e. methods prefixed with _) carry no such restrictions.

6.3 containers

Container classes and enums.

class `repobee_plug.containers.`**BaseParser**
Enumeration of base parsers that an extension command can request.

class `repobee_plug.containers.`**`ExtensionCommand`**
Class defining an extension command for the RepoBee CLI.

class `repobee_plug.containers.`**`ExtensionParser`**
An ArgumentParser specialized for RepoBee extension commands.

class `repobee_plug.containers.`**`HookResult`**
Container for storing results from hooks.

class `repobee_plug.containers.`**`Review`**(*repo*, *done*)

`done`
Alias for field number 1

`repo`
Alias for field number 0

class `repobee_plug.containers.`**`ReviewAllocation`**(*review_team*, *reviewed_team*)

`review_team`
Alias for field number 0

`reviewed_team`
Alias for field number 1

class `repobee_plug.containers.`**`Status`**
Status codes enum.

6.4 corehooks

Hookspecs for repobee core hooks.

Core hooks provide the basic functionality of repobee. These hooks all have default implementations, but are overridden by any other implementation. All hooks in this module should have the *firstresult=True* option to the hookspec to allow for this dynamic override.

class `repobee_plug.corehooks.`**`APIHook`**
Hooks related to platform APIs.

`api_init_requires`()
Return which of the arguments to apimeta.APISpec.__init__ that the given API requires. For example, the GitHubAPI requires all, but the GitLabAPI does not require `user`.

Return type `Tuple[str]`

Returns Names of the required arguments.

`get_api_class`()
Return an API platform class. Must be a subclass of apimeta.API.

Returns An apimeta.API subclass.

class `repobee_plug.corehooks.`**`PeerReviewHook`**
Hook functions related to allocating peer reviews.

`generate_review_allocations`(*teams*, *num_reviews*)
Generate *`ReviewAllocation`* tuples from the provided teams, given that this concerns reviews for a single master repo.

The provided teams of students should be treated as units. That is to say, if there are multiple members in a team, they should always be assigned to the same review team. The best way to merge two teams `team_a` and `team_b` into one review team is to simply do:

```
team_c = apimeta.Team(members=team_a.members + team_b.members)
```

This can be scaled to however many teams you would like to merge. As a practical example, if teams `team_a` and `team_b` are to review `team_c`, then the following *`ReviewAllocation`* tuple, here called `allocation`, should be contained in the returned list.

```
review_team = apimeta.Team(members=team_a.members + team_b.members)
allocation = containers.ReviewAllocation(
    review_team=review_team,
    reviewed_team=team_c,
)
```

Note: Respecting the `num_reviews` argument is optional: only do it if it makes sense. It's good practice to issue a warning if num_reviews is ignored, however.

Parameters

- **`team`** – A list of *`Team`* tuples.
- **`num_reviews`** (`int`) – Amount of reviews each student should perform (and consequently amount of reviewers per repo)

Return type `List`[*`ReviewAllocation`*]

Returns A list of *`ReviewAllocation`* tuples.

6.5 exthooks

Hookspecs for repobee extension hooks.

Extension hooks add something to the functionality of repobee, but are not necessary for its operation. Currently, all extension hooks are related to cloning repos.

class `repobee_plug.exthooks.`**`CloneHook`**

Hook functions related to cloning repos.

`act_on_cloned_repo`(*path*, *api*)

Do something with a cloned repo.

Parameters

- **`path`** (`Union`[`str`, `Path`]) – Path to the repo.
- **`api`** – An instance of `repobee.github_api.GitHubAPI`.

Return type `Optional`[*`HookResult`*]

Returns optionally returns a HookResult namedtuple for reporting the outcome of the hook. May also return None, in which case no reporting will be performed for the hook.

`clone_parser_hook`(*clone_parser*)

Do something with the clone repos subparser before it is used used to parse CLI options. The typical task is to add options to it.

Parameters **`clone_parser`** (`ArgumentParser`) – The `clone` subparser.

Return type `None`

`config_hook`(*config_parser*)
Hook into the config file parsing.

Parameters **`config`** – the config parser after config has been read.

Return type `None`

`parse_args`(*args*)
Get the raw args from the parser. Only called for the clone parser. The typical task is to fetch any values from options added in *`clone_parser_hook()`*.

Parameters **`args`** (`Namespace`) – The full namespace returned by `argparse.ArgumentParser.parse_args()`

Return type `None`

class `repobee_plug.exthooks.`**`ExtensionCommandHook`**
Hooks related to extension commands.

`create_extension_command`()
Create an extension command to add to the RepoBee CLI. The command will be added as one of the top-level subcommands of RepoBee. It should return an *`ExtensionCommand`*.

```
def command(args: argparse.Namespace, api: apimeta.API)
```

The `command` function will be called if the extension command is used on the command line.

Note that the `RepoBeeExtensionParser` class is just a thin wrapper around `argparse.ArgumentParser`, and can be used in an identical manner. The following is an example definition of this hook that adds a subcommand called `example-command`, that can be called with `repobee example-command`.

```
def callback(args: argparse.Namespace, api: apimeta.API) -> None:
    LOGGER.info("callback called with: {}, {}".format(args, api))

@plug.repobee_hook
def create_extension_command():
    parser = plug.RepoBeeExtensionParser()
    parser.add_argument("-b", "--bb", help="A useless argument")
    return plug.ExtensionCommand(
        parser=parser,
        name="example-command",
        help="An example command",
        description="Description of an example command",
        callback=callback,
    )
```

Important: If you need to use the api, you set `requires_api=True` in the `ExtensionCommand`. This will automatically add the options that the API requires to the CLI options of the subcommand, and initialize the api and pass it in.

Return type *`ExtensionCommand`*

Returns A *`ExtensionCommand`*.

6.6 exception

Exceptions for repobee_plug.

exception `repobee_plug.exception.`**`APIImplementationError`**
Raise when an API is defined incorrectly.

exception `repobee_plug.exception.`**`ExtensionCommandError`**
Raise when an :py:class:~repobee_plug.containers.ExtensionCommand: is incorrectly defined.

exception `repobee_plug.exception.`**`HookNameError`**
Raise when a public method in a class that inherits from *`Plugin`* does not have a hook name.

exception `repobee_plug.exception.`**`PlugError`**
Base class for all repobee_plug exceptions.

6.7 name

Utility functions relating to RepoBee's naming conventions.

`repobee_plug.name.`**`generate_repo_name`**(*team_name*, *master_repo_name*)
Construct a repo name for a team.

Parameters

- **team_name** (`str`) – Name of the associated team.
- **master_repo_name** (`str`) – Name of the template repository.

Return type `str`

`repobee_plug.name.`**`generate_repo_names`**(*team_names*, *master_repo_names*)
Construct all combinations of generate_repo_name(team_name, master_repo_name) for the provided team names and master repo names.

Parameters

- **team_names** (`Iterable[str]`) – One or more names of teams.
- **master_repo_names** (`Iterable[str]`) – One or more names of master repositories.

Return type `Iterable[str]`

Returns a list of repo names for all combinations of team and master repo.

`repobee_plug.name.`**`generate_review_team_name`**(*student*, *master_repo_name*)
Generate a review team name.

Parameters

- **student** (`str`) – A student username.
- **master_repo_name** (`str`) – Name of a master repository.

Return type `str`

Returns a review team name for the student repo associated with this master repo and student.

6.8 serialize

JSON serialization/deserialization functions.

`repobee_plug.serialize.`**`json_to_result_mapping`**(*json_string*)

Deserialize a JSON string to a mapping `repo_name: str -> hook_results: List[HookResult]`

Return type `Mapping[str, List[HookResult]]`

`repobee_plug.serialize.`**`result_mapping_to_json`**(*result_mapping*)

Serialize a result mapping `repo_name: str -> hook_results: List[HookResult]` to JSON.

Return type `str`

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index