{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

```
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.

CHAPTER 6

Internal API

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

6.1 apimeta

6.2 pluginmeta

6.3 containers

6.4 corehooks

6.5 exthooks

6.6 exception

6.7 name

6.8 serialize

CHAPTER 7

Indices and tables

- genindex
- modindex
- search