Pautomate Documentation

Release 0.1.0

Ammar Najjar

Apr 29, 2020

Contents:

CHAPTER 1

Pautomate

Automate my boring stuff.

- Free software: MIT license
- Documentation: https://pautomate.readthedocs.io.

1.1 Features

- Uses click under the hood.
- Fetch repos from Gitlab using a personal token.
- Clone the repos if they don't exist.
- Get branches info of repos in a specific directory.
- Reset –hard branches with a flag (-r)
- Run multiple dotnet core commands on different projects in parallel.
- The dotnet command type can be passed as an argument.

1.2 Credits

This package was created originally with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install Pautomate, run this command in your terminal:

```
$ pip install pautomate #TODO
```

This is the preferred method to install Pautomate, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Pautomate can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/ammarnajjar/pautomate
```

Or download the tarball:

```
$ curl  -OL https://github.com/ammarnajjar/pautomate/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

3.1 To use Pautomate in a project

import as:

```
import pautomate
```

3.2 To use Pautomate as a CLI Application

3.2.1 Show help:

Just run:

```
pautomate --help

Usage: pautomate [OPTIONS] COMMAND [ARGS]...

  Console interface for pautomate

Options:
  -t, --target PATH  [.] Target workspace.
  --help             Show this message and exit.

Commands:
  branches    Get branches infos in the local workspace Arguments: - args...
  dotnet      Run dotnet services in parallel via dotnet core CLI Arguments:...
  fetch       Clone/fetch projects from Gitlab using the private token...
```

3.2.2 Commands:

- **branches**:

Get branches informations in the current directory:

```
pautomate -t . branches
```

The target is by default the current directory.

Get the branches informations after resetting hard to head:

```
pautomate beanches -r
```

Careful that this is a **distructive** command, for it uses *git reset –hard* on all the repos treated.

To select only repositories that contatin a specific pattern in them and get thier informations (e.g: *py*):

```
pautomate branches py  # only repositories with "py" in them will be considerd
```

The last set of agruments acts like a filter. More than one argument can also be given, and all will be considerd as filters:

```
pautomate branches py api service  # only repositories with "py" in them will be␣
↪considerd
```

More info can be found using the help command:

```
pautomate dotnet --help
```

- **fetch**:

Clone all repositories on a gitlab isntance using the personal access token. If the repositories is already cloned, it will be fetched again, and the current local branch will be soft reset to the origin branch.

develop branch if exists will also be reset to match *origin/develop*, this guarantees that *develop* stays alway in sync with *origin*.

This command requires some external configurations, and those configurations should be sotred in a *config.json* file in the target directory.

The structure of *config.json* should be like:

```
{
   "gitlab_url": "e.g.: gitlab.com",
   "gitlab_token": "e.g: kjhasd8123hasdz123",
   "ignore_list": ["test", "example"],
}
```

Meanwhile the option *ìgnore_list* is optional, the other two are mandatory to be able to fetch/Clone the repositories from the desired gitlab instance.

More about gitlab personal access tokens can be found in the official documentation.

ignore_list is a list of string patterns to exlclude from fetching/cloning.

To fetch repositories:

```
pautomate fetch
```

To pass a white list pattern:

```
pautomate fetch py demo    # fetch/clone only what has "py" or "demo" in its name
```

More info can be found using the help command:

```
pautomate fetch --help
```

- **dotnet**:

Executes a specific dotnet core command on multiple dotnet core projects in parallel.

A default list can be configured in a *config.json* file in the target directory:

```
{
    "dotnet_projects": ["dotnet_pro1", "dotnet_pro2"]
}
```

So these projects will be looked up then the passed dotnet command will be executed in all of them in parallel.

A process pool will be initialized to conatain the running processes.

All the allowed dotnet commands are supported e.g.:

```
pautomate dotnet run       # run projects in config.json in parallel
pautomate dotnet run -w    # run projects in watch mode
pautomate dotnet test -w   # run test projects in watch mode
pautomate dotnet clean py demo  # dotnet clean only projects that has either "py" or
↪"demo" in its name
pautomate dotnet build demo  # dotnet build only projects that has "demo" in its name
```

More info can be found using the help command:

```
pautomate dotnet --help
```

3.3 Entry Points

There is an extra entry point supported for each command, to make it faster to get the job done. So each command can also be executed in a short form:

```
pautomate fetch    -> fetch
pautomate branches -> branches
pautomate dotnet   -> dnet
```

3.4 Docker

To run using docker:

- build image:

  ```
  docker build --rm -f "Dockerfile" -t pautomate .
  ```

- run the desired entry point:

```
docker run --rm -v $(pwd):/ws:rw -it pautomate --help
docker run --rm -v $(pwd):/ws:rw -it pautomate dotnet --help
docker run --rm -v $(pwd):/ws:rw -it pautomate fetch --help
docker run --rm -v $(pwd):/ws:rw -it pautomate branches --help
```

CHAPTER 4

pautomate

4.1 pautomate package

4.1.1 Subpackages

pautomate.common package

Submodules

pautomate.common.git module

pautomate.common.printing module

pautomate.common.read module

pautomate.common.services module

pautomate.common.timeit module

Module contents

pautomate.git_repos package

Submodules

pautomate.git_repos.branches module

pautomate.git_repos.fetch_gitlab module

Module contents

pautomate.multi_dotnet package

Submodules

pautomate.multi_dotnet.dotnet_exec module

Module contents

4.1.2 Submodules

4.1.3 pautomate.cli module

4.1.4 Module contents

Top-level package for Pautomate.

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/ammarnajjar/pautomate/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

Pautomate could always use more documentation, whether as part of the official Pautomate docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ammarnajjar/pautomate/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *pautomate* for local development.

1. Fork the *pautomate* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pautomate.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pautomate
$ cd pautomate/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 pautomate tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.7. Check https://travis-ci.org/ammarnajjar/pautomate/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ pytest tests.test_pautomate
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Credits

6.1 Development Lead

- Ammar Najjar <najjarammar@protonmail.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

P