braulio Documentation

Release 0.3.0

José María Domínguez Moreno

Apr 06, 2020

Contents

Welcome to this documentation. This tool aims to help with the release process of your project by handling versioning and changelogs for you. Please read the foreword section before installing and using it.

CHAPTER 1

User's Guide

1.1 Foreword

This is a brief introduction about how this tool works and what it can do, so you can know if it is right for your project.

1.1.1 How it works

Braulio walks through all commits of your Git project and classifies them to determine what should be the next version and generate a proper changelog. To do so, it collects only Commits that follow a given *message convention*.

Release steps:

- *Determine the current version*.
- Collect unreleased changes.
- Classify them by type and scope.
- *Determine the new version*.
- Generate the changelog.
- Update files with the new version string.
- Commit and tag.

1.1.2 Version schema

This tool works with the **major.minor.patch** version scheme and most of the features of Semantic Versioning 2.0 are supported. If you use Calendar Versioning or another version schema, this tool is not for you.

It works with final releases as well as **pre-releases**, although you have to configure the *stages of your project first*.

PEP440 dictates how pre-release segments should look. This tool does not enforce anything about the pre-release segment format, but you can configure it to fit PEP440.

1.1.3 Changelog

The changelog is generated using the subject extracted from the commit messages. The output is in ReStructuredText format and can not be customized at this moment.

1.1.4 Project Status

This is still in development, and things may change, but you can give it a try if you want.

1.2 Usage

1.2.1 Releasing a new version

To perform a new release use the **release** subcommand:

```
$ brau release
```

You can let the tool determine the *new version* for you or do a *manual* version release.

Releasing a pre-release version.

To start a pre-release series, Braulio needs a little bit of your help.

Suppose you want the stages of your project to be **dev**, **beta** and **final**, all of them compatible with PEP440. *Go first and set them up*.

Now, let's suppose the current version of your project is `1.6.3` and you want to start working on a new feature, the next version should be `1.7.0`. So to release that version into the **dev** stage run:

```
$ brau relase --minor --stage=dev
```

or using the *–bump* option:

```
$ brau relase --bump=1.7.0dev0
```

From that point, each time you pass **dev** to *–stage*, the numerical part of the pre-release segment will be increased.

```
$ brau release --stage=dev
```

The current version is now `1.7.0.dev1`, if you run it again the version will be `1.7.0.dev2` and so on.

When you are ready to release your first **beta** version, just do it like this:

```
$ brau relase --stage=beta
```

The current version is now `1.7.0b0`. The numerical part of pre-release segments always **starts from 0**.

Finally to release the final version, just run the command without any argument.

```
$ brau relase
```

Braulio knows that the project is currently in a pre-release stage of the version `1.7.0` and will release that final version correctly.

1.2.2 How the current version is found

The application will look for the last **Git tag** that matches *tag_pattern* option, unless *current_version* is provided by the user either via command line or a configuration file.

1.2.3 How the next version is determined

If you follow the *Commit Message Convention* defined for your project, Braulio will be able to know what type of changes introduces each commit and based on that determine what should be the next version.

This table shows what type of commit determines the type of release:

Release type	Commit message metadata
Major release	Commits containing the phrase `BREAKING CHANGE`
Minor release	`feat` type commits.
Patch release	`fix`, `refactor` or any other commit type, including those that doesn't follow the convention.

Right now, only the types `feat` and `fix` are relevant when deciding which version will be the next.

1.2.4 Manual version bump

There are 4 options through the command line interface; *–patch*, *–minor*, *–major* and *–bump*.

Let' supose your current project version is `1.6.3`.

Option	Usage
–patch	`$ brau release --patch` releases to `1.6.4`.
–minor	`$ brau release --minor` releases to `1.7.0`.
–major	`$ brau release --major` releases to `2.0.0`
–bump	`$ brau release --bump=3.0.0` releases to `3.0.0`

1.2.5 Commit Message Convention

Commit messages must have a **label** in a predetermined position. Let's see the default behavior using the example below.:

```
Change the boring music playlist

Here you have a new list of music:
    - La Grange
    - Fuel
    - Sad but true

!fix:music
```

Above, the label is `!fix:music`. By default, a label must follow the format `!{type}:{scope}` and be in the footer. From the previous example the metadata information extracted from the message is as follow.:

- **subject**: Change the boring music playlist
- **type**: fix
- **scope**: music

The subject is important because it appears in the changelog.

The label format and position are customizable via the options *label_position* and *label_pattern*. At this moment, a label can be located only in the **header** or **footer**.

To **customize the label format** use the *placeholders* **{type}**, **{scope}**, and **{subject}**. `{type}` is mandatory while `{scope}` is optional. `{subject}` must be used only when the label is in the message header.

A very popular commit message convention is from the AngularJS project. Here a commit message extracted from their repository.:

```
chore(travis): use Firefox 47

This commit also adds a new capability to the protractor configs that
ensures that all angularjs.org tests run correctly on Firefox. See
SeleniumHQ/selenium#1202
```

For Braulio to understand the above message, we can add the following options to the *configuration file*.

```
[braulio]
label_position = header
label_pattern  = {type}({scope}): {subject}
```

Note that we use **{subject}** because the label is in the header and Braulio needs to know where the subject is to extract it properly. In this case the subject is `use Firefox 47`, the scope is `travis` and the commit type is `chore`.

Important

If the label is located in the footer, **{subject}** must be omitted since the entire header will be used as the subject of the commit message.

Since **{scope}** is optional the next Commit header would be valid:

```
chore(): use Firefox 47
```

In this case, the Commit does not have a specific scope, maybe because the code introduced is too broad.

Breaking changes

At this moment, the only way to let Braulio know that a commit introduces incompatible changes to the codebase is by placing the phrase `BREAKING CHANGE` or `BREAKING CHANGES` somewhere in the body of the message.

No matter what type of commit is specified with the commit label, this phrase will instruct Braulio to perform a major version release.

No matter what type of commit you specify in the commit label, this phrase will instruct Braulio to perform a major version release.

1.2.6 Setting up pre-releases

To support alpha, beta or any other pre-release version, add them under the section `[braulio.stages]` of your project *configuration file*.

Each option under that section is considered a stage of your project and their value must follow the supported version string format (most on that later). Those version formats will be used to parse version strings and serialize them back.

```
[braulio.stages]
dev   = {major}.{minor}.{patch}.dev{n}
beta  = {major}.{minor}.{patch}b{n}
final = {major}.{minor}.{patch}
```

The above indicates that the project release cycle has 3 stages: **dev**, **beta**, and **final** and the order in which they may happen. The name of the options acts as the label of the stage and will be used as the argument for the *–stage* option when needed.

The order in which stages are defined matters because it determines which stages are prior to others. The first defined stages are lower.

You can always release to another stage forward, but not backward. For example, if the current version is `1.5.0beta6`, an attemp to make a dev release `1.5.0.dev0` will fail. If dev and beta were defined in the reverse order, the release would work.

You can bypass a stage, for example, a release from dev (`0.10.0.dev10`) stage to a final stage to (`0.10.0`) will work.

Braulio does not enforce anything about the literal text of the pre-release segments, so you can have something like this:

```
hi = {major}.{minor}.{patch}hello{n}
```

Here another example with alpha and release candidate stages:

```
[braulio.stages]
alpha = {major}.{minor}.{patch}a{n}
rc    = {major}.{minor}.{patch}rc{n}
final = {major}.{minor}.{patch}
```

Finally but not less important, **the final stage should be always included**.

Version string format

The versions string format is defined using *placeholders* and the available ones are:

- **{major}** - Major version part.
- **{minor}** - Minor version part.
- **{patch}** - Patch version part.
- **{n}** - Numerical component that defines the order of releases in a pre-release serie.

The first 3 are always mandatory and must be separated by a dot character.:

```
{major}.{minor}.{patch}
```

Following then, any word or character can be present. **{n}** must be at the end of the string pattern. The next examples are all valid.:

```
# alpha release
{major}.{minor}.{patch}a{n}

# Another alpha release style
{major}.{minor}.{patch}a{n}

# This have a dot (.) after the patch part
{major}.{minor}.{patch}.dev{n}

# Withou a dot (.)
{major}.{minor}.{patch}dev{n}
```

1.2.7 About placeholders

This tool uses string patterns in many of the options it has, but they are not Regular Expressions.

Instead, it uses placeholders surrounded by curly braces `{}` as the Python Format String Syntax. Anything that is not contained in braces is treated as literal text.

They are used not only to render new strings but also to extract information.

For example, *tag_pattern* is used to find all Git tags that represent a released version and requires the placeholder `{version}`. If the pattern is `release-{version}`, `release-2.0.1` will match but `released-2.0.1` won't because the literal part is not equal.

The extracted placeholder information in the above example is `2.0.1`. When a new version is released, `2.2.0` for example, the new tag name will be rendered to `release-2.2.0`.

1.3 Configuration

Most of the options that let you configure Braulio's behavior, are available through the command line tool or a *Config file* with a few exceptions.

The options provided through the CLI have precedence over those specified in the configuration file.

1.3.1 Config file

Currently, only the file **setup.cfg** can be used to configure the application. All the options must be under the section `[braulio]`. There is a special section: `[braulio.stages]` which is used solely to configure the stages of the project.

A config file would look like this:

```
[braulio]
commit = False
Tag = False
confirm = True

[braulio.stages]
alpha = {major}.{minor}.{patch}a{n}
beta  = {major}.{minor}.{patch}b{n}
final = {major}.{minor}.{patch}
```

1.3.2 Options

Next, we have a table with all the options for the release subcommand. If an option is not available through an input method, the cell will be empty.

CLI	config file	Descriptions
–major		Major version bump.
–minor		Minor version bump.
–patch		Patch version bump.
–bump		Bump to a given version arbitrarily.
–commit / –no-commit	commit	Enable/disable release commit
–message	message	Customizes commit message.
–tag / –no-tag	tag	Enable/disable version tagging.
–changelog-file	changelog_file	Specify the changelog file.
–label-position	label_position	Where the label is located in the commit message.
–label-pattern	label_pattern	Pattern to identify labels in commit messages.
–tag-pattern	tag_pattern	Pattern for Git tags that represent versions
–current-version	current_version	Manually specify the curren version.
–stage		Select a stage where to bump
–merge-pre		Merge pre-release changelogs.
-y	confirm	Don't ask for confirmation
files (argument)	files	Don't ask for confirmation
–help		Show this message and exit.

bump

CLI	Config File	Default
`--bump`		

Takes a valid version string and bump the project version.

major

CLI	Config File	Default
`--major`		

Perform a major release bumping the major part of the current version.

minor

CLI	Config File	Default
`--minor`		

Perform a major release bumping the major part of the current version.

patch

CLI	Config File	Default
`--patch`		

Perform a major release bumping the major part of the current version.

current_version

CLI	Config File	Default
`--current-version`	current_version	

Determines the current version of the project. If this option is present in the configuration file, it will be updated on each new release.

tag_pattern

CLI	Config File	Default
`--tag--patern`	tag_pattern	`v{version}`

Parse and render Git tag names.

It is used to find Git tags that mark a release, as well as to render tag name for new releases.

The pattern string must have the *placeholder* `{version}`, which determines where a version string is located in a tag name.

Examples:

The tag pattern `version{version}` would match `version1.0.0`. The tag pattern `release-{version}` would match `release-1.0.0`.

As stated above, any time a new version is released, the same pattern will be used to render the new Git tag name.

label_position

CLI	Config File	Default
`--label-position`	label_position	`footer`

Determines where the commit analyzer must look for commit labels. The available values are **header** and **footer**.

label_pattern

CLI	Config File	Default
`--label-pattern`	label_pattern	`!{type}:{scope}`

The format of label inside commit messages. This uses the next *placeholders* to extract metadata information.:

- **{type}**: The type of the commit (fix, feat, chore, etc). Required.

- **{scope}**: The scope where a commit belong. Optional.
- **{subject}**: The subject of the message. Required when the label is located in the header.

commit

CLI	Config File	Default
`--commit/--no-commit`	commit	True

Enable/disable commit of the changes produced by a version bump. If this is enabled, it will commit only the changelog file and the files provided through the *files* option.

tag

CLI	Config File	Default
`--tag/--no-tag`	tag	True

Enable/disable a release tag after a version bump.

message

CLI	Config File	Default
`--message`	message	"Release version {new_version}"

If the release commit is enabled, this is used for the message.

This is a template string containing replacement fields. The available fields are **{new_version}** and **{current_version}**. `{new_version}` is mandatory, while `{current_version}` is optional.

changelog_file

CLI	Config File	Default
`--changelog-file`	changelog_file	

Path to the changelog file.

files

CLI	Config File	Default
`files (argument)`	files	

List of files to update with a new version string.

Note that in the case of the CLI this is a positional argument and must be place at the end of the command.

```
$ brau release --bump=4.0.0 file1.py file2.py folder/file3.py
```

Each file path must be separated by an space.

Through a configuration file, each file path must be in a new line like the example belog.

```
[braulio]
files =
    file1.py
    file2.py
    folder/file3.py
```

stage

CLI	Config File	Default
`--stage`		

Determines in what stage a new release must be made.

stages

CLI	Config File	Default
	[braulio.stages]	`final = {major}.{minor}.{patch}`

Only available through a configuration file, this determines the stages of a project development cycle.

By default the only stage defined is **final**, which must always present:

```
[braulio.stages]
final = {major}.{minor}.{patch}
```

For more information, read the *Setting up pre-releases* section.

1.4 Installation

1.4.1 Stable release

To install braulio, run this command in your terminal:

```
$ pip install braulio
```

This is the preferred method to install braulio, 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.

1.4.2 From sources

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

You can either clone the public repository:

```
$ git clone git://github.com/mbarakaja/braulio
```

Or download the tarball:

```
$ curl  -OL https://github.com/mbarakaja/braulio/tarball/master
```

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

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

1.5 History

1.5.1 0.3.0 (2018-08-22)

Bug Fixes

- release
  - Abort when a lower version is passed to –bump
  - Stop aborting when user inputs No to confirmation prompt
  - Ensure –bump works with versions without minor and patch parts.
  - Validate tag_pattern value
- git - Fix Tag’s __repr__ and __str__ methods

Features

- release
  - Add –merge-pre option
  - Add –stage option
  - Support pre-release versions
  - Add option to customize the commit message
  - Add option to specify the current version
  - Add support to custom git tag names
  - Add support to custom commit message conventions
- cli - Add –version option to output current version

1.5.2 0.2.0 (2018-07-25)

Bug Fixes

- changelog - Fix release markup being inserted in the wrong place

Features

- release
  - Show useful info while running release subcommand
  - Add support to custom change log file names
  - Support version string update on selected files
- init - Add interactive config and changelog files creation

1.5.3 0.1.0 (2018-07-13)

Features

- release
  - Add –no-commit and –no-tag options
  - Add options for manual version bump