Bulk Documentation

Release 0.4.2

paul@colomiets.name

Jul 11, 2018

Contents

Contents:

CHAPTER 1

Configuring Bulk

Contents:

1.1 Overview

Bulk's configuration file is usually `bulk.yaml` in the root if your project but can be overrident by `-c` or `--config` for most subcommands.

This is yaml config parsed by quire so you can use most of its features there.

Configuration file consists of a declaration of minimum supported version of bulk and three sections, here is an example:

```
minimum-bulk: v0.4.5

versions:
- file: setup.py
  regex: ^\s*version\s*=\s*["']([^"']+)["']
- file: your_module/__init__.py
  regex: ^__version__\s*=\s*["']([^"']+)["']

metadata:
  name: your-app
  short-description: A great app in python
  long-description:
    A very great app in python

repositories:
- kind: debian
  suite: bionic
  component: your-app
```

All sections are optional if you don't want to use some of the functionality here. In particular:

1. `versions` needed if you want to keep project version in source code in multiple places and want to update it using bulk
2. `metadata` is a package metadata, if you don't build `.deb` package you don't need it
3. `repositories` is a repository metadata, if you don't have debian/ubuntu repository you don't need it.

1.2 Versions

Versions section help you with bookkeeping a version in your application, here is the sample of versioning for python application

```
versions:
- file: setup.py
  regex: ^\s*version\s*=\s*["']([^"']+)["']
- file: your_module/__init__.py
  regex: ^__version__\s*=\s*["']([^"']+)["']
```

You might also add an example to your readme and keep version in documentation updated too:

```
versions:

- file: setup.py
  regex: ^\s*version\s*=\s*["']([^"']+)["']

- file: your_module/__init__.py
  regex: ^__version__\s*=\s*["']([^"']+)["']

- file: doc/conf.py
  regex: ^version\s*=\s*u?["']([^"']+)["']
  partial-version: ^\d+\.\d+  # no patch version

- file: doc/conf.py
  regex: ^release\s*=\s*u?["']([^"']+)["']

- file: README.rst
  regex: pip\s+install\s+your-module==(\S+)
```

Options:

`file` Filename to search version in, relative to project directory (usually a directory that contains `bulk.yaml`)

`files` A list of files to search. This is useful if you can use same regex in multiple files.

> **Note:** Neither existence of `file` or any one in `files` is enforced. if you make a typo file will be silently skipped. Always use `bulk check-version` after modifying rules.
>
> On the upside is that you can use same `bulk.yaml` for many similar projects and versions that aren't present will be skipped.

`regex` A regular expression that matches version. It must contain a single capturing group (i.e. `a (parenthised expression)`) for capturing actual version. Regex can match only on a single line.

The expression shouldn't be too strict and should not try to validate the version number itself. I.e. if version is quoted anything inside the quotes should be considered version, if it isn't anything to next white space or newline is okay.

Too strict version pattern risk to be either to replace `1.2.3` to `1.2.4` in `1.2.3-beta.1` keeping beta suffix, or to skip `1.2.3-beta.1` line in a file without updating it because it doesn't match.

If `regex` matches multiple times all matching lines are treated as version number. Also multiple entries with the same file and different rules can be configured.

`partial-version` A regular expression that allows to select only portion of version number. Few examples:

1. `^\d+\.\d+` – selects major.minor version but not patch
2. `-.*$` – selects `-alpha`, `-beta.1`, `-31-g12bd530` or any other pre-release suffix in version number.

`block-start`, `block-end` Marks block where to find version number in.

For example, in `Cargo.toml` version number is in the `[package]` section and named `version`, whereas `version=` in other case may denote version of other things like pedependencies. So we use this:

```
- file: Cargo.toml
  block-start: ^\[package\]
  block-end: ^\[.*\]
  regex: ^version\s*=\s*"(\S+)"
```

You can specify single file multiple times in versions section. Which effectively means you can fix version in multiple different sections.

`multiple-blocks` (default `false`) By default bulk stops scanning this file for this rule on the first `block-end` after `block-start`. If this setting is set to `true` searches for the next `block-start` istead. This option does nothing if no block defined.

1.3 Package Metadata

Package information is stored in `metadata` section in `bulk.yaml`. Here is an example:

```
metadata:
  name: your-app
  short-description: A great app in python
  long-description:
    A very great app in python. You can use it to do
    amazing things
  depends: [python3]
```

Options:

`name` Package name used to name a `.deb` file

`short-description` Short description of the package as shown in package search results and in other places. It should be a one-liner

`long-description` Long description of the package. Usually shown in GUI tools as a part of package detail.

`depends` List of package dependencies. It can consists of any expression allowed in debian packages. But note if you need different dependencies for different packages built (i.e. for different ubuntu distributions) you need to use different `bulk.yaml` configs and specify ones explicity to `bulk pack`.

1.4 Repositories

When using bulk it's common to track multiple repositories using single config. Here is an example:

```
repositories:

- kind: debian
  suite: bionic
  component: your-app-testing
  keep-releases: 1000

- kind: debian
  suite: bionic
  component: your-app
  keep-releases: 1
  match-version: ^\d+\.\d+\.\d+$
```

This keeps 1000 releases in testing repository. And just one release in stable repository. Where stable release has strict semantic version and all releases are included in testing repository (including stable). Non-stable releases themeselves are probably versioned with `git describe` yielding versions like this: `1.2.3-34-gde103b3`.

Options:

`kind` Kind of the repository. Only `debian` is currently supported.

`suite` Suite of the repository. For ubuntu it's usually a release codename such as `xenial` or `bionic`.

`component` Component of the repository. Common convention is that it's a application name (so technically you can put multiple applications in the same repository). Also it may include modifier like `-testing` or `-stable`.

`keep-releases` Number of releases of the package to keep in this repository. By default all releases are kept (i.e. it's never cleaned up). Usual debian tools keep exactly one package.

It's also a good idea to keep two repositories: `your-app` with `keep-releases: 1` and `your-app-stable` with `keep-releases: 100` which keep older packages. The index of the first repository is smaller and faster to download and the latter can be used to downgrade. Note: repositories share a pool of packages so `.deb` file itself isn't duplicated for two repositories.

`match-version` Only add version matching this regex to the repository.

There are two good usecases for the feature:

1. Sort out testing and stable versions (as in example above)
2. Use a single `bulk repo-add` command to add packages for every distro. This works by append something like `+bionic1` suffix to a package version and add a respective `match-version` for that distribution.

`skip-version` This is the same as `match-version` but is a negative filter. If both are matched `skip-version` takes precedence.

`add-empty-i386-repo` (default `false`) When building `amd64`-only repo also add an empty index for `i386` counterpart. This is needed to prevent errors on `apt update` on systems which are configured to fetch both 64bit and 32bit versions of packages.

For now it's known that ubuntu precise (12.04) default install only has this problem. So since precise reached its end of life this option is deprecated.

CHAPTER 2

Version Bookkeeping

Bulk can be used to sync version of your application to various places in code.

2.1 Basics

Bulk uses regular expressions to find versions in some file. For example, here is how we track versions in typical python project:

```
versions:

# There is usually a version in setup.py
- file: setup.py
  # this isn't 100% correct as version can end in different quote or
  # there might be few version parameters in a file, but this is good
  # enough for many projects, other projects might need to tweak matcher
  regex: ^\s*version\s*=\s*["'](^"']+)["']


# Also it's a good idea to put library version into
# a __version__ attribute of the module itself
- file: your_module/__init__.py
  regex: ^__version__\s*=\s*["'](^"']+)["']
```

Put it in `bulk.yaml` and now you can find out version with:

```
> bulk get-version
1.3.5
```

Yes, the first time you've written `setup.py` and `__init__.py` you needed to put version yourself. This is usually handled by project boilerplate.

2.2 Releasing a Project

If you obey **semantic versioning** in the project version run one of:

```
> bulk bump --breaking -g
> bulk bump --feature -g
> bulk bump --bugfix -g
```

The commands above will increment a major, minor or patch version of your version number, commit the changes with a comment of `Version bumped to v1.3.6` and create an annotated tag `v0.3.6` by starting an editor and showing you changes since previous tag. You can opt-out commit and tag creation by omitting `-g` which is equivalent of longer `--git-commit-and-tag`.

You can also use `-1`, `-2` and `-3` which increment the specific component of version. Technically they are are equivalent to above except when version is zero-based `0.x`.

Note: in case of `0.x` versions the version numbers are shifted. I.e. if you have two zeros numbers `0.0.x` any bump with increment a single version. If you have `0.x.y` number second component will increment with both `--breaking` and `--feature`. This is how many existing tools handle semver. Use `-1`, `-2` if in doubt or to switch from `0.x` versions to `1.x`.

For **date-based versioning** use:

```
> bulk bump -dg
```

This will force your version to something like `v180317.0`. If you will subsequently run this command on the same day you will get `v180317.1` and so forth.

Note: The date here is UTC to avoid issues with different people releasing in different timezones.

Another way to update is to use `set-version`:

```
> bulk set-version v1.3.5-beta.1
./your_module/__init__.py:1: (v1.3.5 -> v1.3.5-beta.1) __version__ = '1.3.5-beta.1'
./setup.py:6: (v1.3.5 -> v1.3.5-beta.1)       version='1.3.5-beta.1',
```

This is useful to set some pre-release version as you see in example because we don't have a command-line flag for that or in case you have different version format or just want to skip version number for some reason.

2.3 Building a Pre-Release Project

Everyting above assumes that version is stored in source code and commited to git. Which is true for many tools. But you don't want to commit version for a prerelease version of application. We have a nice command for this use case too:

```
> bulk with-version v1.3.6-pre4 your-build-command
1.3.5 -> 1.3.6-pre4
[ .. output of your-build-command .. ]
1.3.6-pre4 -> 1.3.5
```

This runs build with correct version and ensures that when build is complete you will get no version change in git status.

Since the common case is using `git describe` for actual version we have a shortcut for that:

```
> bulk with-git-version your-build-command
1.3.5 -> 1.3.5-4-gd923e59-dirty
[ .. output of your-build-command .. ]
1.3.5-4-gd923e59-dirty -> 1.3.5
```

(the `-dirty` here means you have modified git-tracked files locally)

Note: The `git describe` command is not strictly semver-compatible. I.e. the version `x.y.z-n` is treated as lower than `x.y.z` and you're supposed to use `x.y.z+n` for that. But for now we decided to stick to what `git describe` provides for now. We may provide an option to fix that in future, in the meantime you can use `with-version`.

2.4 Other Commands

To check if version number is fine (consistent) run:

```
> vagga bulk check-version
setup.py:6: (v1.3.5)       version='1.3.5',
trafaret_config/__init__.py:1: (v1.3.5) __version__ = '1.3.5'
```

It shows you files and lines where version number is present and will fail if there is no version at all or version is inconsistent between multiple files.

Note: it will **not** show you files and lines which are present in config file but has no version number found. So when adding an entry in `bulk.yaml` you should run `check-version` and make sure the actual entry exists in the file.

To fix inconsistent version run:

```
> vagga bulk set-version v1.3.5 --force
setup.py:6: (v1.3.4 -> v1.3.5)       version='1.3.5',
trafaret_config/__init__.py:1: (v1.2.3 -> v1.3.5) __version__ = '1.3.5'
```

Same restriction for not found version as for `check-version` applies here.

CHAPTER 3

Q & A

3.1 Why version number and file existence is optional?

Sometimes we want to put and edit version number in generated files: lock-files, code generated things and other.

Since entries in `bulk.yaml` are almost never modified it's much easier to check once after editing a file than to learn rules of what is strict and what isn't.

Here is just one example, when it is useful. Here is how we configure bulk in rust projects:

```
- file: Cargo.toml
  block-start: ^\[package\]
  block-end: ^\[.*\]
  regex: ^version\s*=\s*"(\S+)"

- file: Cargo.lock
  block-start: ^name\s*=\s*"project-name"
  regex: ^version\s*=\s*"(\S+)"
  block-end: ^\[.*\]
```

The important part is that we *must* update `Cargo.lock` so that `bulk set-version/incr-version/bump -g` works fine *(we modify ''Cargo.lock'' together with ''Cargo.toml'' and commit in the same commit, if we don't do that lockfile is update on next build and needs to be commited after)*.

But we also want to be able to run bulk with absent lockfile (in case we don't commit it into a repository) or if we want cargo to rebuild it from scratch.

CHAPTER 4

Indices and tables

- genindex
- search