Mnglt Documentation

Release 0.1.0

Pierre Fernique

June 20, 2016

Contents

Summary

Version 0.2.0
Status
Author see *Authors* section
ChangeLog see *Change-log* section
License CeCILL-C (see *License* section)

MngIt aims at managing redundant information within sofware. This information collection is done using a central configuration file and using information provided by Version Control System (VCS) repositories. Then, information is dispatched in various files in order to minimize redundancy. **MngIt** also aims at providing commands to ease development using web-based repository hosting services.n Given precise workflows, the process of using VCS with web-base repository hosting services can be eased for developers.

CHAPTER 1

Documentation

1.1 User guide

1.1.1 Managing redundant information within software

We here assume that you are using the **Git** Version Control System (VCS) to develop a repository denoted `<reponame>`.

```
$ mkdir <reponame>
$ cd <reponame>
$ git init
$ ls -a
. .. .git
```

Using a configuration file, **MngIt** aims at dispatching redundant information in various files that can be considered as mandatory in repositories. The configuration file named `.mngit.yml` is created at the root of `<reponame>` using the `mngit config` command.

```
$ mngit config
$ ls -a
. .. .git .mngit.yml
```

These informations are then dispatched in the repository using the `mngit update` command. This configuration file is formatted according to the **YAML** human friendly data serialization standard. This configuration file is incrementally filled using commands that are presented in the remainder of this documentation.

The `software` field

The `.mngit.yml` contains a field `software` that contains 2 sub-fields:

- The name of the software (denoted by `<name>`).

  Warning: `name` sub-field must be formatted according to restructuredText markup syntax.

- A short description of the software (denoted by `<shortdesc>`).

  Warning: `desc` sub-field must be formatted according to restructuredText markup syntax.

These informations are provided as follows (see Lst. 1.1)

```
$ mngit software --name <name> --desc <shortdesc>
```

Lst. 1.1: Status of the `.mngit.yml` file

```
software:
   desc: <shortdesc>
   name: <name>
```

The `about` field

The `.mngit.yml` contains a field `about` that contains some additional information concerning the software:

- An email for corresponding authors (denoted by `<email>`)
- A long description of the software (denoted by `<longdesc>`).

> **Warning:** `desc` sub-field must be formatted according to restructuredText markup syntax. Empty lines are not allowed.

These informations are provided as follows (see Lst. 1.1)

```
$ mngit about --contact <email> --desc <details>
```

Lst. 1.2: Status of the `.mngit.yml` file

```
software:
   desc: <shortdesc>
   name: <name>
 about:
   contact: <email>
   desc: <longdesc>
```

The `version` field

Like most VCSs, **Git** has the ability to tag specific points in history as being important. Using the `mngit version` command ensures that the version will be automatically updated from the **Git** history at each update (see Lst. 1.4).

```
$ mngit version
```

Note: As illustrated on Lst. 1.3, once the `mngit version` command has been executed, a default version (0.1.0 for `semantic version` plugin) is added in the `software` field.

Lst. 1.3: Status of the `.mngit.yml` file

```
about:
    contact: <email>
    desc: <longdesc>
software:
    desc: <shortdesc>
    name: <name>
    version: 0.1.0
version:
    plugin: semantic
```

To detect versions, **MngIt** considers the last tag corresponding to certain patterns. To handle different versioning patterns, the `version` plugin manager is used. Currently, **MngIt** provide the `semantic version` plugin that corresponds to the semantic versioning design (e.g. tags considered are matching the `'(v)?([0-9]*).([0-9]*).([0-9]*)(-.*)?'` regular expression).

```
$ git add -A
$ git commit -m 'Add MngIt configuration file'
...
$ git tag v0.2.0
$ mngit update
```

Lst. 1.4: Status of the `.mngit.yml` file after the execution of the `git tag` command

```
about:
  contact: <email>
  desc: <longdesc>
software:
  desc: <shortdesc>
  name: <name>
  version: 0.2.0
version:
  plugin: semantic
```

The `authors` field

If you consider that a file (assumed to be `AUTHORS.rst`) listing repository authors is mandatory, you can use the `mngit authors` command to ensure an automatic update of this file.

```
$ mngit authors
```

This file is updated from authors that are retrieved from the **Git** log.

```
$ mngit update
$ ls -a
. .. AUTHORS.rst .git .mngit.yml
```

An author is identified by its name and email. For each author a score is computed according to the plugin given to the `authors` plugin manager. Currently, **MngIt** provide a `commit authors` plugin that compute the score of authors as their number of commits. Authors are written according to a specified `format` sub-field in decreasing order in the file named accoording to the `basename` sub-field (see Lst. 1.5) .

Warning: `format` sub-field must contains substitutions for `name`, `email` and `score` that are identified by braces.

Lst. 1.5: Status of the `.mngit.yml` file

```
software:
  desc: <shortdesc>
  name: <name>
  version: 0.1.0
about:
  contact: <email>
  desc: <longdesc>
version:
```

```
  plugin: semantic
authors:
  basename: AUTHORS.rst
  format: '* {name} <{email}> ({score})'
  plugin: commit
```

The `license` field

If you consider that a file (assumed to be `LICENSE.rst`) containing repository license is mandatory, you can use the `mngit license` command to ensure an automatic update this file. This will also update all your license headers in source code files. For creating these field you must first precise the license to consider (see Lst. 1.6).

```
$ mngit license --plugin CeCILL-C
```

Lst. 1.6: Status of the `.mngit.yml` file

```
software:
  desc: <desc>
  name: <name>
  version: 0.1.0
version:
  plugin: semantic
authors:
  basename: <authors>
  format: '* {name} <{email}> ({score})'
  plugin: commit
license:
  basename: LICENSE.rst
  plugin: CeCILL-C
  width: 78
```

How the `LICENSE.rst` and source code files are updated is controlled by the `plugin` sub-field that is used by the `license` plugin manager to select corresponding plugin. For source code license headers, the width of the header is controller by `width` sub-field.

```
$ mngit update
$ ls -a
. .. AUTHORS.rst .git LICENSE.rst .mngit.yml
```

The `restructuredtext` field

In order to dispatch informations in repository reStructuredText files using the `mngit update` command, you can use the `mngit restructuredtext` command. This command require a list of target files within which reStructuredText substitutions are defined between `..  MngIt` gards:

- `|NAME|` can be used to get the `name` sub-field of `software` field.
- `|BRIEF|` can be used to get the `desc` sub-field of `software` field.
- `|VERSION|` can be used to get the `version` sub-field of `software` field.
- `|DETAILS|` can be used to get the `desc` sub-field of `about` field.
- `|AUTHORSFILE|_` can be used to get a link to the file identified by the `basename` sub-field of `authors` field.
- `|LICENSENAME|` can be used to get the `plugin` sub-field of `license` field.

- |LICENSEFILE|_ can be used to get a link to the file identified by the basename sub-field of license field.

Warning: These substitutions are available only if corresponding fields and sub-fields are in the .mngit.yml file.

To define targetted files, use this command as follows (see Lst. 1.7)

```
$ mkdir doc
$ touch doc/index.rst
$ touch README.rst
$ ls -R
.:
AUTHORS.rst doc LICENSE.rst README.rst

./doc:
index.rst
$ mngit restructuredtext --target README.rst doc/index.rst
```

Lst. 1.7: Status of the .mngit.yml file

```
about:
  contact: <email>
  desc: <longdesc>
authors:
  basename: AUTHORS.rst
  format: '* {name} <{email}> ({score})'
  plugin: commit
license:
  basename: LICENSE.rst
  plugin: CeCILL-C
  width: 78
restructuredtext:
  plugin: default
  target:
  - README.rst
  - doc/index.rst
software:
  desc: <shortdesc>
  name: <name>
  version: 0.3.0
version:
  plugin: semantic
```

Note: For now, we only consider the **Git** VCS but features could be added to in order to accept the **Subversion** VCS.

Note: This documentation only illustrate command purposes. For more detailed information use the -h option (e.g. mngit -h, mngit config -h...)

1.1.2 Commands to ease development using *GitHub*

We here assume that you are using the Git Version Control System (VCS) and *GitHub* as web-based repository hosting service. We present a workflow to contribute to an official repository denoted <reponame> in an organization (or

user account) denoted <orgname>. The workflow describe here is mainly inspired from OpenAlea and Virtual Plants public development workflow. Considering this workflow, **MngIt** provides some commands to ease the process.

Fork and clone a repository

To fork and clone the repository <reponame> of the <organame> organization, we recommend to use the mngithub clone command instead of *GitHub* interface and **Git** commands. To perform these steps, mngithub clone uses the **PyGithub** package to access *GitHub* interface in *Python*. Your gitHub credentials (<username> and <password>) are therefore required.

```
$ mngithub clone
Username for 'https://github.com': <username>
Password for 'https://pfernique@github.com': <password>
```

The, you must enter:

- the organization name (<organame>).

```
Enter an organization name: <organame>
```

> **Note:** In place of an organization, you can give a *GitHub* user.

- the repository name (<reponame>).

```
Enter a repository name: <reponame>
```

> **Warning:** The <reponame> must exists in the <organame> organization.

This command:

1. Fork the <reponame> of the <organame> organization into your account.

   > **Note:** If you already forked the <reponame> it will not be forked one more time.

2. Clone it on your disk at your current location within the <reponame> directory.

   > **Warning:** If your fork of the <reponame> repository is not named <reponame>, Its name will be used in place of <reponame> for the directory that contains the clone.

   > **Note:** By default, the clone is performed using SSH remote url. If you prefer to use the HTTPS remote url, use the url argument of the mngithub clone command:

   ```
   $ mngithub clone --url=https
   ```

3. Add the upstream remote that refers to the repository on the <organame> organization.

   > **Note:** By default, the origin remote refers to the repository on your account.

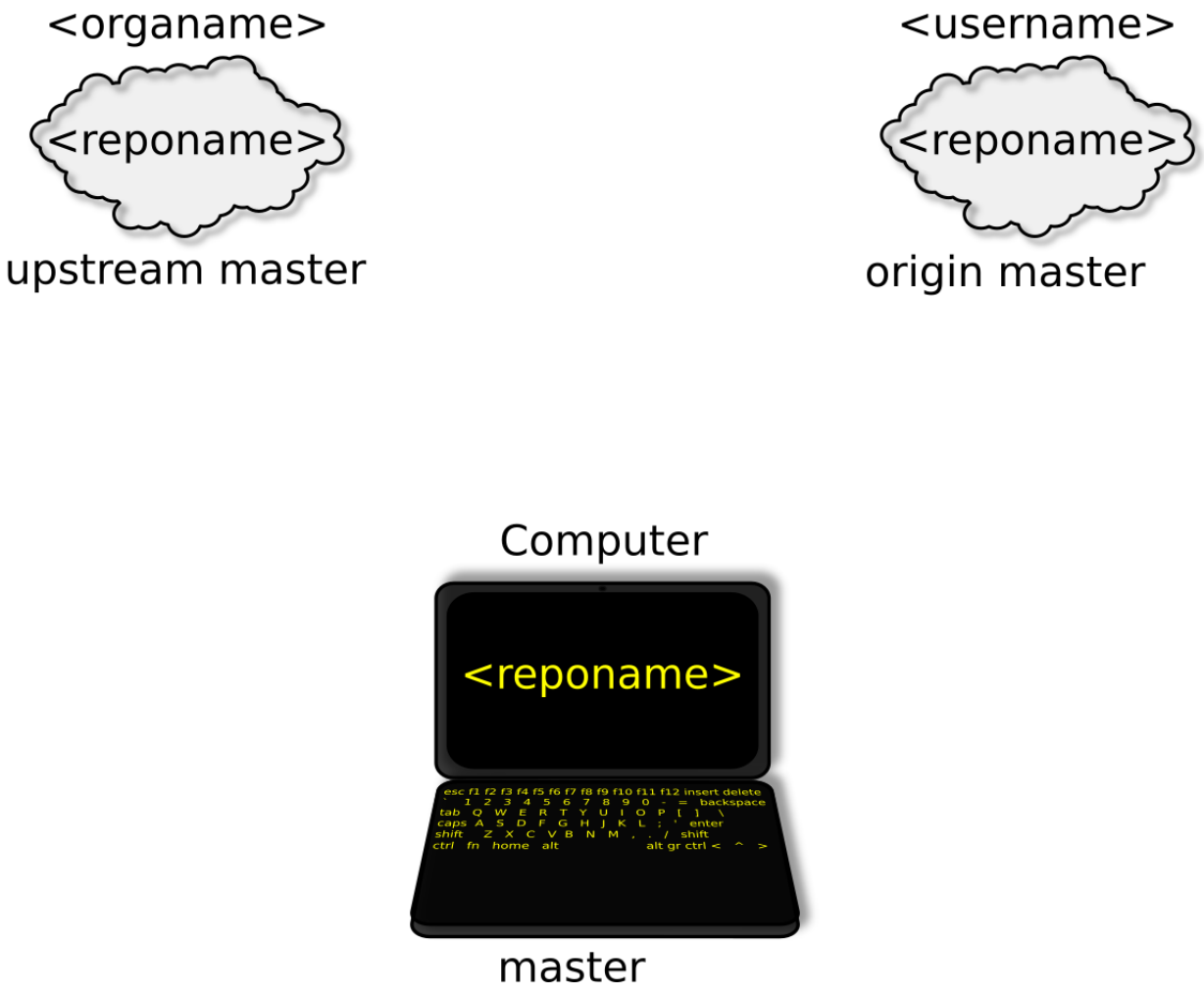


Fig. 1.1: Repository status after fork and clone of a repository.
The repository is named `<reponame>`. The left hand cloud represents the repository on the `<organame>` organization *GitHub* account. The right hand clound represents the forked repository on your `<username>` *GitHub* account. The computer represents the cloned repository on your computer.

Note: Cloning a repository using **Git** only adds the `origin` remote. The principale value added of the `mngithub clone` command is to add the `upstream` remote. For example, this enable you to compare your local branch to both remote branches using:

- for the remote branch on the `<organame>` *GitHub* account,

```
$ git diff upstream/master
```

- for the remote branch on your `<username>` *GitHub* account,

```
$ git diff origin/master
```

Or, since `origin` is chosen by default,

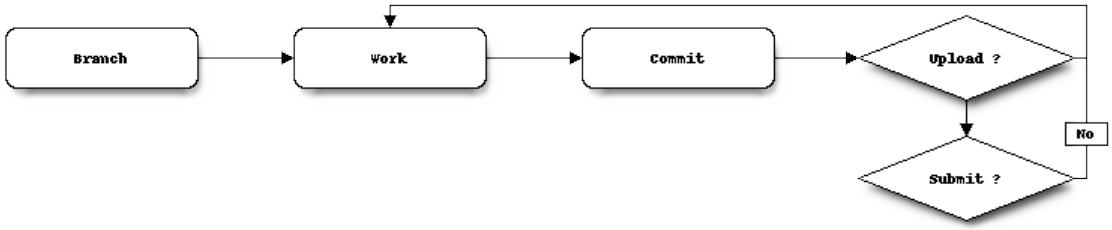
```
$ git diff master
```

Contribute to a repository

When using **Git** you should constently keep in mind the following warning:

> **Warning:** Never work on master, always on a branch

In order to contribute to the `<reponame>` repository of the `<organame>` organization we therefore recommand to follow the following workflow.



Name	Description
Branch	See *Create a development branch* section. In order to enable code review from mainteners, the development must be short (i.e. one branch for one task such as new feature, bug fix...).
Work	See *Work on your modifications* section. In order to benefit from tools developped by mainteners and ensure code quality, the development must respect some guidelines.
Commit	See *Commit your modifications* section. Commits are snapshots of the repository. There are useful in particular for versionning software or create backups.
Upload ?	See *Upload your modifications* section. In order save your modifications into your personal repository, you should upload them. Otherwise, you can continue to add commits.
Submit ?	See *Submit your modifications* section. In order to integrate your modifications to the official repository, you must submit your modifications that will be integrated by organization mainteners.

Note: In the following we assume that you forked the official repository in your personal account and cloned it according to previous recommendations (see Fork and clone a repository).

Create a development branch

In order to enable code review from mainteners, the development must be short (i.e. one branch for one task such as new feature, bug fix...). Moreover, the more the development cycle is long, the more you will risk to have conflicts.

The process of development branch creation is detailed in Fig. 1.2 but `mngithub branch` commands do this for you.

```
$ mngithub branch <branchname>
```

Note: Please choose an explicit name `<branchname>` for your branch.

Fig. 1.2: Steps of the development branch creation.
Repositories of the same color are synchronized. Before the creation of your development branch, all three repositories are not synchronized. In:

1. Your local `master` branch is synchronized with the `upstream master` branch.

```
git checkout master
git pull upstream master
```

2. Your remote `origin master` branch is synchronized with your local `master` branch.

```
git push
```

3. Since all your master branches are synchronized, the local `<branchname>` branch is created

```
git branch <branchname>
git checkout <branchname>
```

   Or equivalently

```
git checkout -b <branchname>
```

4. Then, the remote `origin <branchname>` branch is created in order to enable the uploading of future modifications into your `<username>` *GitHub* account.

```
git push --set-upstream origin <branchname>
```

Warning: After the execution of the `mngithub branch` command, your local repository has switched on the `<branchname>` branch.

```
$ git branch
master
* <branchname>
```

But there are, at this point, no differences between the `master` and `<branchname>` branches.

```
$ git status
On branch <branchname>
nothing to commit, working directory clean
```

Note: Once this step is done, refers to the *workflow* to continue.

Work on your modifications

Note: Once this step is done, refers to the *workflow* to continue.

Commit your modifications

The commit of modifications with **Git** is quite different from **Subversion**. In particular, **Git** will not consider that your local `<branchname>` branch differs from `origin <branchname>` branch until you committed your modifications (see Fig. 1.3).

Fig. 1.3: Effect of **Git** commits
Until you committed your modifications (1.), **Git** will not consider that your local <branchname> branch differs from origin <branchname> remote branch.

Note: While master and origin master are still synchronized, it is assumed that some work from other developpers has been integrated into the upstream master. There are therefore two different versions of master branches at the end of this step.

Warning: The commit of modifications do not implies the upload of these modifications. The branches <branchname> and origin <branchname> are therefore no more synchronized.

The *repository index* In **Git**, the *repository index* notion is primordial (see the this post for more details). In short, files in the *repository index* are files that would be committed to the repository if you used the git commit command. However, files in the *repository index* are not committed to the repository until you use the git commit command. Therefore, in order to commit your modifications you must first build the *repository index* using file additions and removals. For this step the git status, git add and git rm commands are your friends:

git status Tells you what files:

- have been added to the *repository index*,
- exists in the working tree but are not in the *repository index*,
- have different contents between the working tree and the *repository index*.

git add <pathspec> Add the <pathspec> file to the repository index.

Warning: Contrarily to **Subversion**, with **Git** the git add command must be performed not only for adding new files but also for modified files. By default no file is added in the index.

For more details, refers to the **Git** manual (git add --help).

git rm <pathspec> Remove the <pathspec> file from the working tree and the index. For more details, refers to the **Git** manual (git remove --help).

Note: If you do not want to remove the <pathspec> file from you working tree but only in the *repository index* use git rm --cached <pathspec> instead.

Note: Since the incremental addition or removal of files can be tidious, the commands git add -A can be of most interest. This command will also add files that were created. Therefore in order to add only relevant files, the .gitignore file is of most importance (see create).

The Git Commit Once the index is build as desired, it must be committed in order to make another snapshot of the repository. This is done using the git commit command. If you leave off the -m option, this command open your favorite editor (see ../configuration) where you can construct a message associated to the commit. Two commits are distinguished:

Backup & service commits These commits are not corresponding to particular development stages and can be used when uploading is a neccessity. For example these commits arise when a developper wants to:

- Remotly save his developments.
- Use a service (see create).

For this type of commits, please use the `git commit -m "<shortdesc>"` command where `<shortdesc>` is a short summary of the commit. This summary should be less that 50 characters.

Developement commits The commits are all commits not considered as backup. Please avoid the usage of the `-m` option and produce a nice commit message using the follwing steps (the reader can refer to the A Better Git Commit message to more informations):

- The first line should be a short summary. Referencing the bug number or the main accomplishment of the change (e.g "Fixes issue #8976). This is the title of your commit and should be less than 50 characters.
- Then a line break.
- Followed by a longer detailed description about the things that changed. This section is a really good place to explain what and why. You could cover statistics, performance wins, roadblocks, etc. The text should be wrapped at 72 characters.

Note: If you want to add to your index deleted or modified files when committing, you can use the `-a` flag. The command

```
git commit -a
```

is used for automatically staged files that have been modified and deleted, but new files you have not told **Git** about are not affected. In this fact this command is different from the commands

```
git add -A
git commit
```

that will also add new files.

Note: Once this step is done, refers to the *workflow* to continue.

Upload your modifications

Once you have committed your modifications, you can upload them in your `<username>` *GitHub* account using the `git push` command (see Fig. 1.4).

Fig. 1.4: Steps of the development branch creation.
Repositories of the same color are synchronized. Before the creation of your development branch, all three repositories are not synchronized. In:

Submit your modifications

Fig. 1.5: Steps of the development branch creation.
Repositories of the same color are synchronized.

Prepare your pull-request Before submitting your modifications, you must recover changes from `upstream master` remote branch in your local `master` branch

```
git checkout master
git pull upstream master
```

and upload the changes in your `origin master` remote branch

```
git push
```

Then, you must rebase your local development branch with your local `master` branch.

```
git checkout <branchname>
git rebase master
```

If conflicts occur, fix conflicts for each file and finish rebase

```
git rebase --continue
```

Note: Any file modified when fixing conflicts should be added using the `git add <pathspec>` command.

If anything has gone wrong, you can abort reabase

```
git rebase --abort
```

Fig. 1.6: Steps of the development branch creation.
Repositories of the same color are synchronized

Create your pull-request On github interface, select your branch `<branchname>` and click on pull-request (see this post for more details).

Warning: You must see the following message: "Able to merge. These branches can be automatically merged". If it's not the case, the `upstream master` has probably diverged. You must therefore turn back to previous step (see *Prepare your pull-request* section).

If all steps described in the workflow are respected, your branch is clean and mainteners have absolutely nothing to do to integrate your work (except to review your changes) and so it will certainly be integrated.

Fig. 1.7: Steps of the development branch creation.
Repositories of the same color are synchronized

Integrate your pull-request

Note: Once your branch is integrated in the `upstream master`, it is recommanded to to delete your branch:

- On your local repository,

```
git checkout master
git branch -d <branchname>
```

- On your personal repository,

```
git push origin --delete <branchname>
```

Warning: Once this step is done, refers to the *workflow* to continue.

See *StatisKit <statiskit.readthedocs.io/en/latest/maintener/index.html>*'s maintener reference guide

1.2 Reference guide

mngit.config.**dump_config**(*repository*, *config*)

mngit.config.**init_config**(*repository*, ***kwargs*)
Create a MngIt configuration file

Parameters

- *name* (str) - Name of the software
- *brief* (str) - A brief description of the software

mngit.config.**load_config**(*repository*)

mngit.languages.**get_language**(*basename*)

mngit.load_rst_default.**load_restructuredtext**(*repository*, *filepath*, *config*)

CHAPTER 2

Authors

- Pierre Fernique <pfernique@gmail.com> (250)

CHAPTER 3

Change-log

CHAPTER 4

License

MngIt is distributed under the CeCILL-C license.

CHAPTER 5

Indices

- genindex
- modindex

m

D

G

I

L

M