git-flow Documentation

Release 1.0

Johan Cwiklinski

Mar 05, 2020

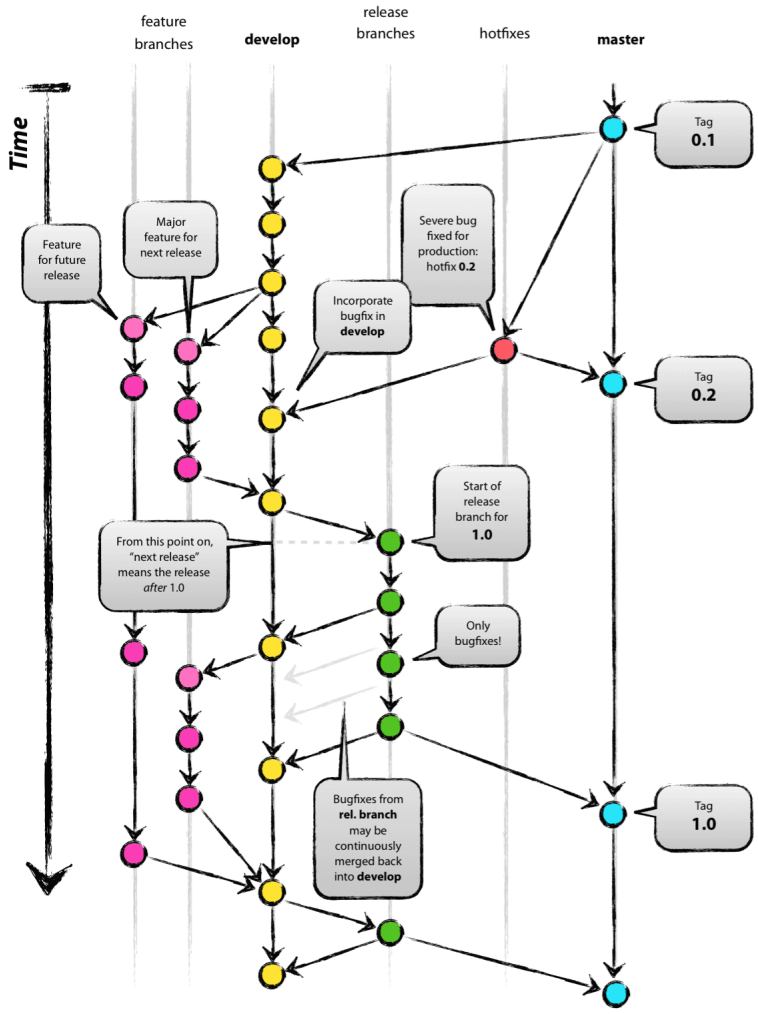
Contents

We've decided to use the git-flow workflow for the GLPI plugins we maintain. This documentation will show some use cases, which commands to use and when to use them.

CHAPTER 1

Presentation

git-flow is a branching model, which come along with some documentation, and a git plugin to add command line facilities.



Keep in mind that it is just designed to get something standardized; all the background use standard git commands, you can achieve "manually" everything git-flow propose. It is just simplier to use, and it prevents to use the incorect branch, or to forget about merging somewhere.

It is not the goal of the present documentation to list pros and cons of this model, we'll note that it is not designed to get long running support branches, it has been once something that would have been implemented; but it has not been done.

According to the semantic versionning rules:

- you'll add `features` only for *major* or *minor* versions,
- you'll `release` *major* or *minor* versions,
- you'll `hotfix` *patch* versions.

1.1 Conventions

The present documentation assumes that:

- a `$` sign will precede each command line instructions,
- any term beetween `()` before the `$` sign is the name of the current branch,

- all is driven from the command line (I do not use any git GUI anyways).

1.2 Pre-requisites

In order the get the commands available; you'll have to install the git-flow git plugin.

Most of Linux distributions have it in their repositories (so `yum install git-flow` or `apt-get install git-flow` would do the trick) or you can follow the installation instructions provided on the project wiki.

Many GIT software are aware of gitflow, or can be if you install a simple plugin; check their respective documentation.

If you use command line, there are numerous ways to get usefull informations displayed in your prompt. While this is not a pre-requisite, it can help you save time!

Fig. 1: As an example, the ZSH git prompt I use

1.3 Working with Github

Each project will have a main repository hosted on Github. Even if you are part of the core developers, you will use the main repository only to push changes on the `develop` and `master` branches. All other branches will be created on a fork (use the eponym button at the top of the project - see below) you will create on your account.

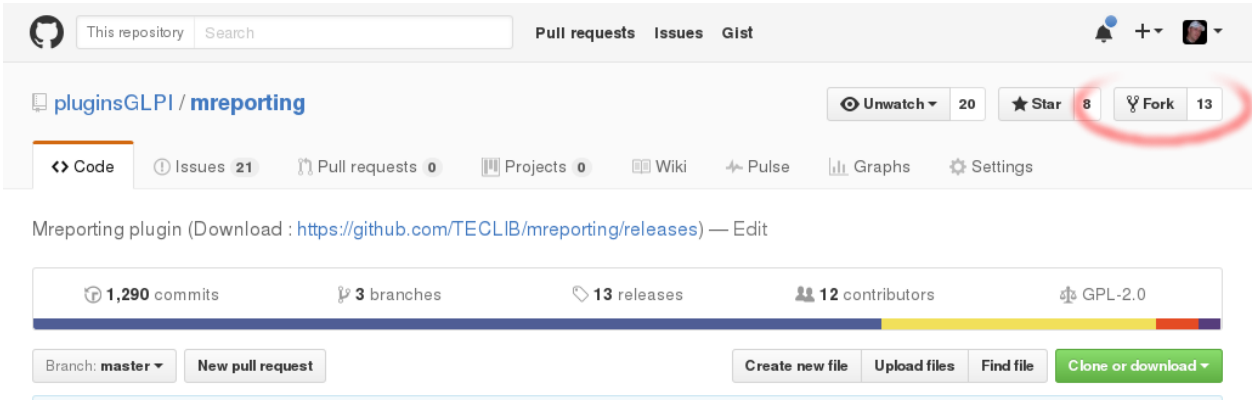


Fig. 2: The fork button

From the main repository you've cloned the project to, add a new remote, let's say naming as your github username (name does not matter, just remember what you choose, and stay consistent accross projects). Replacing `{github_username}` with your own username, run the following:

```
$ git remote add {github_username} git@github.com:{github_username}/mreporting.git
```

All branchs you will create that must be reviewed will be pushed to your fork.

1.4 Initialization

Initializing git-flow is quite simple, just clone the repository, go to the `master` branch and run:

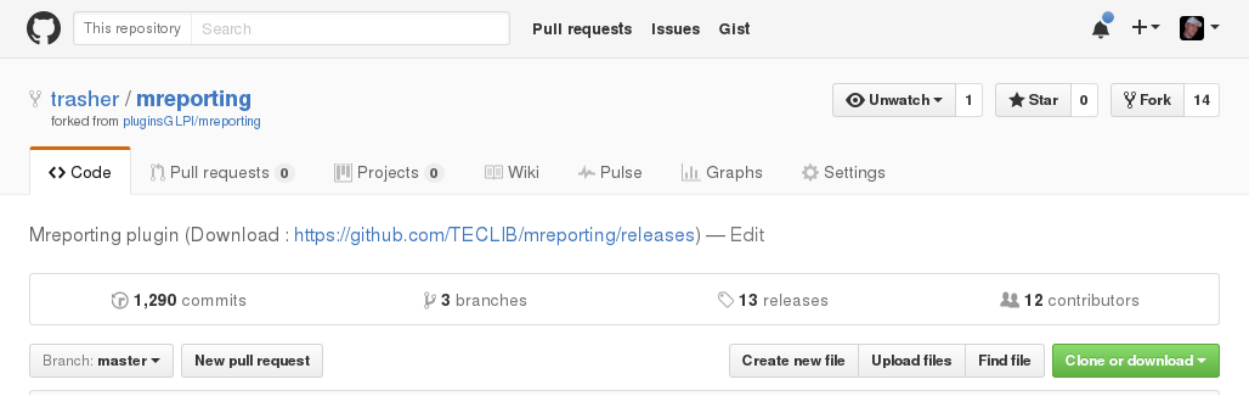


Fig. 3: The forked repo on my personnal account

Note: When you clone a git repository, the default branch will be checkout. In most cases, it will be `master`, but double check.

```
(master) $ git flow init
```

You can assume the default answer is correct for all questions. If the `develop` branch already exists, it will be used, the process will create it otherwise.

1.5 Not finished process

On some occasions, a git-flow command may not finish (in case of conflict, for exemple). This is really not a problem since its fully managed :)

If a git-flow process is stopped, just fix the issue and run the same command again. It will simply run all tasks remaining.

Note: To be sure everything worked as expected, always take a close look at the ouptut!

1.6 merge vs rebase

Should I merge or should I rebase? Well, it's up to you!

Warning: Even if both solutions can be used, and you can choose one or another on some cases; always remember that a `rebase` can be destructive! Keep that in mind.

In facts, you can repair a rebase issue, but only on your local workspace (using `reflog`). Note this is really something you should not use if you're not a git expert ;)

I do not want to feed any troll; both have pros and cons. My advice would be to avoid merge commits when it is not required. I'll try to explain some common cases, and the way I do manage them with the few following examples…

You work on a *feature*; all that ends once squashed into one only commit. By default, the git-flow process will add your commit on the `develop` branch and will add an (empty) merge commit also. This one is really not required, it only make history less readable. If the merge commit is not empty, this begin to be more complicated; you probably miss a `git flow feature rebase` somewhere.

Conclusion: use **rebase**

You've added a hotfix, again one only commit. git-flow will create merge commits as well. For instance, I'm used to keep those commits, this is a visual trace in the history of what has been done regarding bug fixes.

Conclusion: use **merge**

You've finished a *feature*, just like someone else... But other side changes have already been pushed to remote `develop`. If you run `(develop) $ git push`, you will be informed that you cannot push because remote has changed.

I guess many will just run a `(develop) $ git pull` in that case, that will add a merge commit in your history. Those merge commits are really annonying searching in history, whether they're empty or not. As an alternative, you can run `(develop) $ git pull --rebase`, this will prevent the merge commit.

Conclusion: use **rebase**

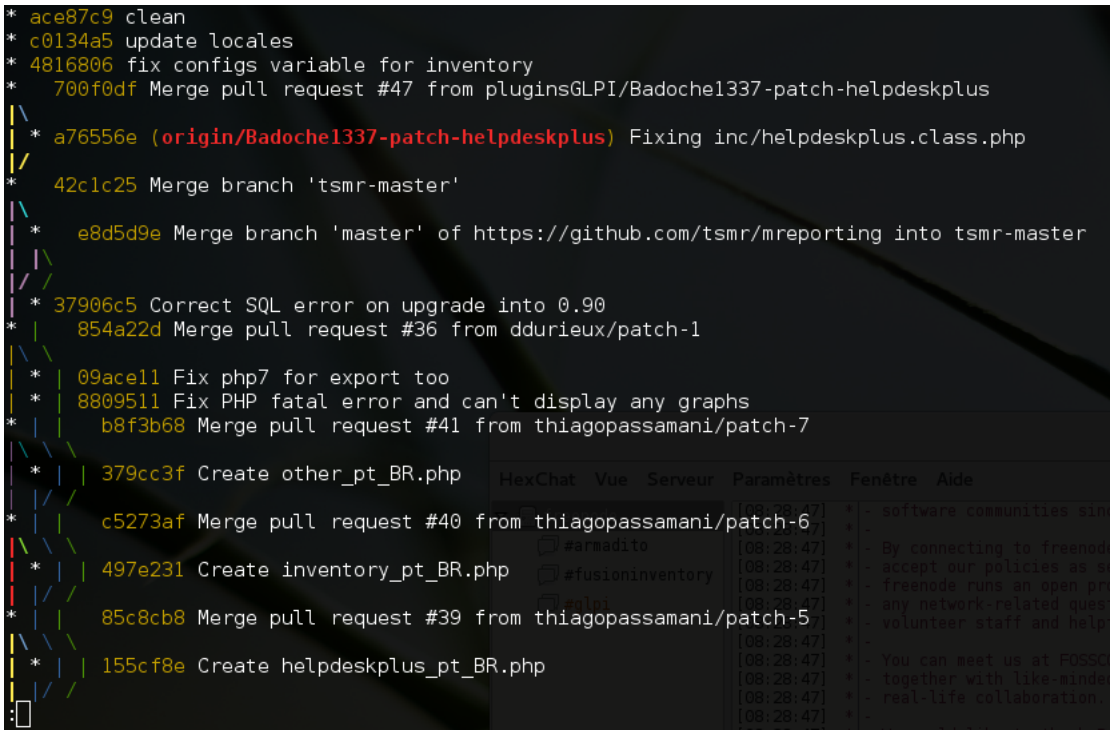


Fig. 4: An example history (from the mreporting plugin).

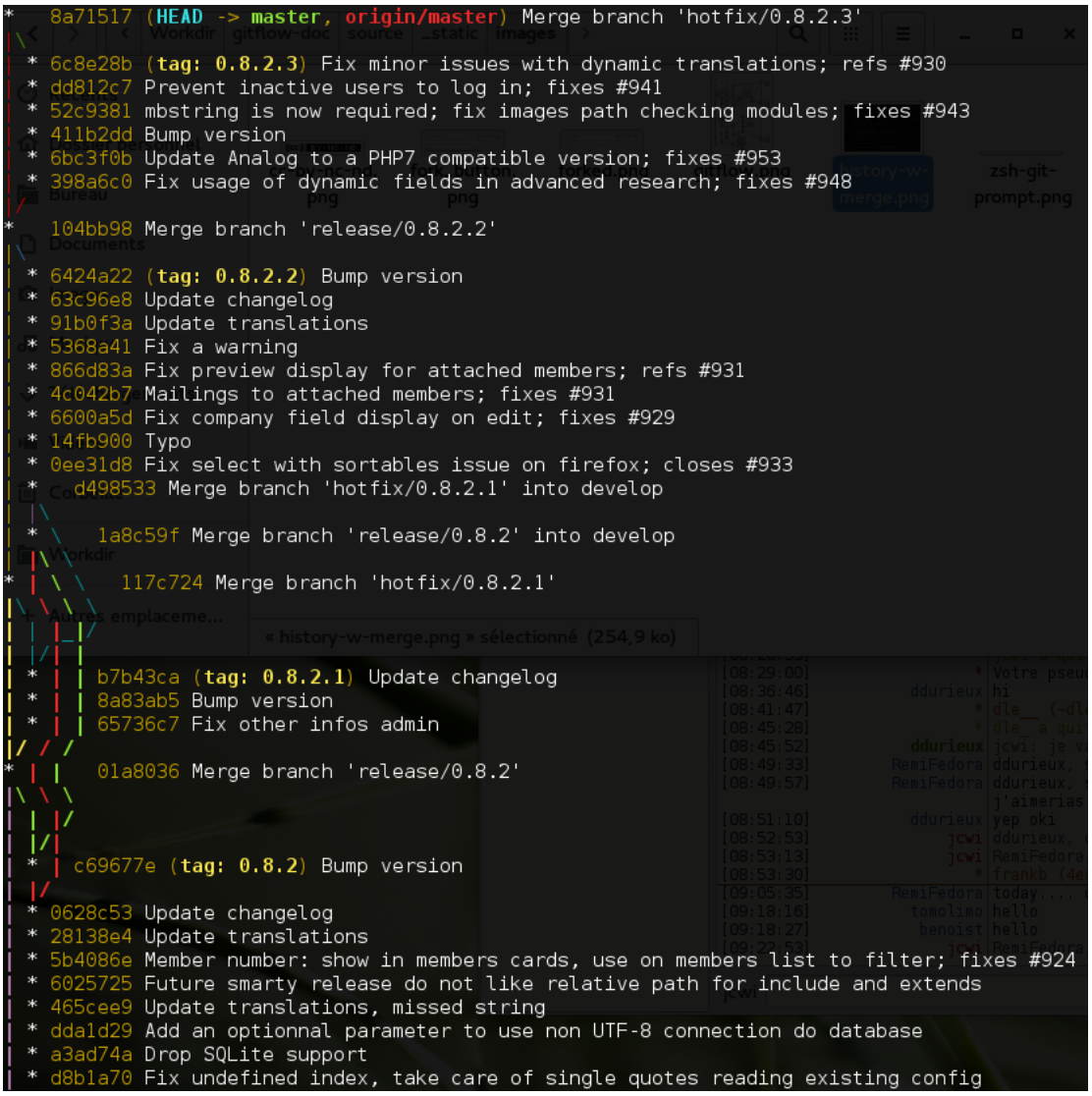


Fig. 5: Another example history (from the Galette project).

CHAPTER 2

Features

Warning: Be carefull your `develop` branch is up-to-date before starting or finishing a feature! Even if you can quite easily fix that ;)

This possibility will be used to implement new features in the project. Features are designed to be created from the `develop` branch, and reintegrated in the `develop` branch as well.

Note: You can have as many features as you want on a project; you'll see at usage than working with too many features can be a kind of nightmare; as well as long running ones ;)

2.1 Creation

The name of the feature is up to you, choose something simple and short, describing what you are doing. To start a feature named *my-great-feature* you'll use:

```
$ git flow feature start my-great-feature
```

This will automatically do the following:

1. create a new branch named `feature/my-great-feature` from the `develop` branch,
2. checkout the `feature/my-great-feature` branch.

So, yes, you're ready to go! Just hack, commit, push, ... You're on a branch, you can do what you want (well... almost!).

As stated in *Working with Github*, you will use your fork to push new branches. You'll achieve that running:

```
(feature/my-great-feature) $ git push -u {github_username} feature/my-great-feature
```

2.2 Lifetime

Sometimes, nothing happened on the `develop` branch until you finish your feature, and you'll have nothing to take care of.

But sometimes, some other features have been added, or some bugs have been fixed... You'll have to keep you feature branch up-to-date. Considering your `develop` branch has been updated (you always keep your `develop` updated, don't you? :p):

```
(feature/my-great-feature) $ git flow feature rebase
```

This will rebase you feature branch on top of the develop; it sounds just like you've just created your feature right now and it applies your commit one by one on the top. Explaining rebasing is out of the scope of the current documentation, but you'll find many resources on it.

2.3 Pull Request

Your feature has been finished, it must now be reviewed before being merged. Push last changes to your fork, go to your github fork page, select your branch and clik "New pull request" button.

You can provide aditionnal details if any, submit, and wait for another developer to review your changes! Once accepted, go back to your local copy, and see the paragraph below.

2.4 Finishing

Once you're done, and your PR has been accepted, you can clean up a bit your branch history, squashing your commits to prevent keeping commit messages like "oops, I did it again!". Assuming your working copy is on the feature branch, you'll then run:

```
(feature/my-great-feature) $ git flow feature finish
```

This will do the following:

1. merge branch `feature/my-great-feature` into `develop`,
2. ask you for a commit message (default will be "Merge branch 'feature/my-great-feature' into develop")
3. delete branch `feature/my-great-feature`

For the the second step, you can just save the message as it; if you've squashed your commits, you can remove the merge commit simply using:

```
(develop) $ git rebase -i
```

Or not, *it's up to you* :)

Finally, push the `develop` branch for remote repository to be updated! And then you're done, the *my-great-feature* has reached `develop` and will be part of the next release! Congratulations o/

Do not forget to remove remote `feature/my-great-feature` branch:

```
$ git push {github_username} :feature/my-great-feature
```

CHAPTER 3

Hotfix

Warning: Be carefull your `master` branch is up-to-date before starting a *hotfix*, and both your `master` and `develop` branches are up-to-date before finishing it!

You will use *hotfix* to fix bugs against the latest stable release of the project, no matter it was a *major, a *minor* or another *patch*.

Note: You can have **only one** *hotfix* at the same time!

3.1 Creation

The name of the hotfix must be the release it will become. If the latest release was *1.3.2*; you'll want to create a *1.3.3* hotfix using:

```
$ git flow hotfix start 1.3.3
```

This will automatically do the following:

1. create a new branch named `hotfix/1.3.3` from the `master` branch,
2. checkout the `hotfix/1.3.3` branch.

3.2 Lifetime

Just like *Features*, you will have nothing to do if there were no changes on the `master` branch since you've created your *hotfix*.

If something has changed in the `master`, that means another *hotfix* has already been done, which also means that the version you are using is probably incorrect now. In that case, you will have to:

- rename your *hotfix* branch,
- update the code

Assuming the *1.3.3* version has been released from another *hotfix*, you will work on the *1.3.4* version:

```
(hotfix/1.3.3) $ git branch -m hotfix/1.3.4
(hotfix/1.3.4) $ git rebase -i master
```

3.3 Pull Request

Your *hotfix* has been finished, it must now be reviewed before being merged. Push last changes to your fork, go to your github fork page, select your branch and clik "New pull request" button.

You can provide aditionnal details if any, submit, and wait for another developer to review your changes! Once accepted, go back to your local copy, and see the paragraph below.

3.4 Finishing

Warning: Before running the commands to end your *hotfix*, make sure that:

- your `master` branch is up-to-date
- no other *hotfix* using the same version number has been merged (use `git tag | sort -V`)

Warning: You **have to use Git command line, and not Github facilities** to finish the hotfix!

Finishing a *hotfix* is as simple as:

```
$ git flow hotfix finish 1.3.4
```

This will:

- Merge changes into the `master` branch,
- Create a `1.3.4` tag,
- Merge changes into the `develop` branch,
- Remove your local `hotfix\1.3.4` branch.

Once your *hotfix* has been finished; you'll have to push `master`, `develop` and `tags` and also remove remote `hotfix/1.3.4` branch:

```
(master)  $ git push
(master)  $ git push --tags
(master)  $ git checkout develop
(develop) $ git push
          $ git push {github_username} :hotfix/1.3.4
```


BY-NC-ND

CHAPTER 4

Releases

Warning: Be carefull your `develop` branch is up-to-date before starting a release, and both your `master` and `develop` branches are up-to-date before finishing it!

You will use the release feature to publish new *minor* or *major* versions, but not *patches*. It is designed to begin a new fresh release from the `develop` branch.

Note: You can have several releases on a project; but honestly, I cannot find a use case where it should really be used. It's up to you ;)

4.1 Creation

Just as *hotfixes*, the branch name must be the version it will become. Let's say we want to release a new *minor* `1.4.0`:

```
$ git flow release start 1.4.0
```

This will automatically do the following:

1. create a new branch named `release/1.4.0` from the `develop` branch,
2. checkout the `release/1.4.0` branch.

4.2 Lifetime

Warning: Until it's finished, you can still add new *hotfixes* or *features* (anyways, if a new feature must reach your release, you've a planning issue ;)).

But keep in mind **nothing will not reach your release branch until you do something!**.

Most of the time, your release branch should have a quite short lifetime; and changes should be very light comparing your `develop`. As an example, on several project I own (or I've owned); the *release* branch was created to update the changelog if any, add the release date, and eventually bump the version.

This kind of branch may be used for testing purposes also.

Sometimes, you can also just create a *release* to finish it immediately without doing any changes... :-)

If a new *hotfix* has been added, than you'll have to get it back to your release branch. To know how to proceed, you'll have to determine if something else has changed; because you probably do not want a feature finished after you decide to release to be backported.

Note: Remember that you should always prefer to merge or cherry pick rather than report changes manually; that would cause conflicts while finishing.

In the simpliest case, nothing else has changed in your `develop`, just update it and run:

```
(release/1.4.0) $ git merge develop
```

If there were other changes, it may be a bit more complex. You can either `cherry-pick` the fix commit, or use advanced git possibilities of `merge` command (such as merging a specific range of commits, for example); refer to the *Git documentation*.

4.3 Pull request

If you've just created the *release* to bump the version, it is not mandatory to open a pull request. On the other hand, if you've made fixes, you'll have to.

If you're on the second use case, push last changes to your fork, go to your github fork page, select your branch and clik "New pull request" button.

You can provide aditionnal details if any, submit, and wait for another developer to review your changes!

Once accepted, or if you do not need a PR, go back to your local copy, and see the paragraph below.

4.4 Finishing

Warning: Before running the commands to end your *release*, make sure that:

- your `master` and `develop` branches are up-to-date
- no other tag using the same version number has been created (use `git tag | sort -V`)

Warning: You **have to use Git command line, and not Github facilities** to finish the release!

Finishing a *release* is as simple as:

```
$ git flow release finish 1.4.0
```

This will:

- Merge changes into the `master` branch,
- Create a `1.4.0` tag,
- Merge changes into the `develop` branch,
- Remove your local `release\1.4.0` branch.

Once your *release* has been finished; you'll have to push `master`, `develop` and `tags` and also remove remote `release/1.4.0` branch (if any):

```
(master)  $ git push
(master)  $ git push --tags
(master)  $ git checkout develop
(develop) $ git push
          $ git push {github_username} :release/1.4.0
```

CHAPTER 5

References

A few references and links relative to git-flow.

- model explained (original author)
- *git extension <https://github.com/nvie/gitflow>* (original author)
- a quick tutorial
- another tutorial, by Atlassian
- a cheat sheet
- a french video tutorial
- stack overflow questions

Some links about git itself:

- official website
- the ProGit book you can view online or download for free; and that has also been printed
- Atlassian tutorials
- an interactive cheat sheet (I find this one really usefull to understand all interactions and "levels"
- a Stanford publication with several translations

5.1 Tools

Well, I do use vim to code, and the git extension to manage my worflow; even if some vim plugins exists; I do not use them.

Anyways, if you use a tool that implements git-flow natively or with a plugin; feel free to open a PR on the current doc!


BY-NC-ND