git-flow Documentation

Version 1.0

Johan Cwiklinski

mars 05, 2020

Table des matières

1 Présentation **3**
- 1.1 Conventions . . . 4
- 1.2 Pré-requis . . . 5
- 1.3 Travailler avec Github . . . 5
- 1.4 Initialisation . . . 5
- 1.5 Processus non terminé . . . 6
- 1.6 merge vs rebase . . . 6

2 Features **9**
- 2.1 Création . . . 9
- 2.2 Cycle de vie . . . 10
- 2.3 Pull Request . . . 10
- 2.4 Finalisation . . . 10

3 Hotfix **13**
- 3.1 Création . . . 13
- 3.2 Cycle de vie . . . 13
- 3.3 Pull Request . . . 14
- 3.4 Finalisation . . . 14

4 Releases **15**
- 4.1 Création . . . 15
- 4.2 Cycle de vie . . . 15
- 4.3 Pull request . . . 16
- 4.4 Finalisation . . . 16

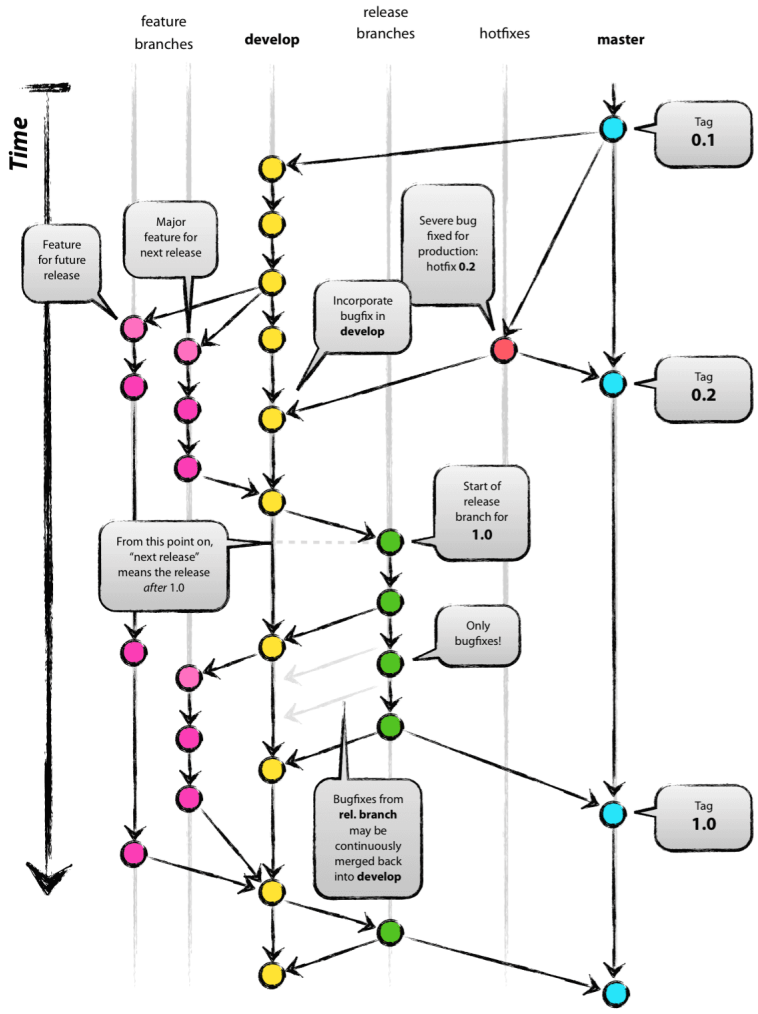
5 Références **19**
- 5.1 Outils . . . 19

Nous avons décidé d'utiliser le workflow git-flow pour les plugins GLPI que nous maintenons. La présente documentation montrera certains cas de figure, quelles commandes utiliser et quand y avoir recours.

CHAPITRE 1

Présentation

git-flow est un modèle de branche, qui est fourni avec de la documentation, et un plugin git pour ajouter des commandes qui facilitent le travail.



Gardez en tête qu'il s'agit de standardiser ; des commandes git standard sont utilisées an arrière-plan, vous pourriez obtenir le même résultat « manuellement » qu'avec git-flow. C'est juste plus simple à utiliser, et ça évite d'utiliser la mauvaise branche, ou d'oublier de merger quelque part.

Le but de la présente documentation n'est pas de lister les pour et les contre de ce modèle, on notera simplement qu'il n'est pas prévu pour fournir des branches de support à long terme, c'est quelque chose qui avait été évoqué mais qui n'a finalement jamais été implémenté.

D'après les règles de versionnage sémantiques :

— vous ajouterez des `features` pour les versions *majeures* et *mineures* uniquement,
— vous créerez des `release` pour des versions *majeures* ou *mineures*,
— vous créerez des `hotfix` pour les versions *patch*.

1.1 Conventions

La présente documentation part du principe que :

— un signe `$` précèdera toute instruction en ligne de commande,
— tout terme entre `()` avant le signe `$` représente le nom de la branche courante,
— tout est fait depuis la ligne de commande (je n'utilise pas d'interface graphique pour git de toutes façons).

1.2 Pré-requis

Pour que les commandes soient disponibles, vous devrez installer le plugin git git-flow.

La majorité des distributions linux le fournissent dans leur dépôts (donc `yum install git-flow` ou `apt-get install git-flow` devrait faire l'affaire) ou vous pouvez suivre les instructions d'installation depuis le wiki du projet.

Beaucoup de logiciels GIT supportent git-flow, ou le peuvent par le biais de plugins ; consultez les documentations respectives.

Si vous utilisez la ligne de commande, il existe de nombreux moyens pour afficher des informations utiles dans le prompt. Bien que ce ne soit pas un pré-requis, cela peut vous faire gagner du temps !

Fig. 1 – Par exemple, le prompt git ZSH que j'utilise

1.3 Travailler avec Github

Chaque projet aura un dépôt principal hébergé sur Github. Même si vous faites partie des développeurs principaux, vous n'utiliserez le dépôt principal que pour pousser les modifications des branches `develop` et `master`. Toutes les autres branches devront être créées sur un fork (utilisez le bouton éponyme en haut de la page du projet - voir ci-dessous) que vous allez créer sur votre compte.

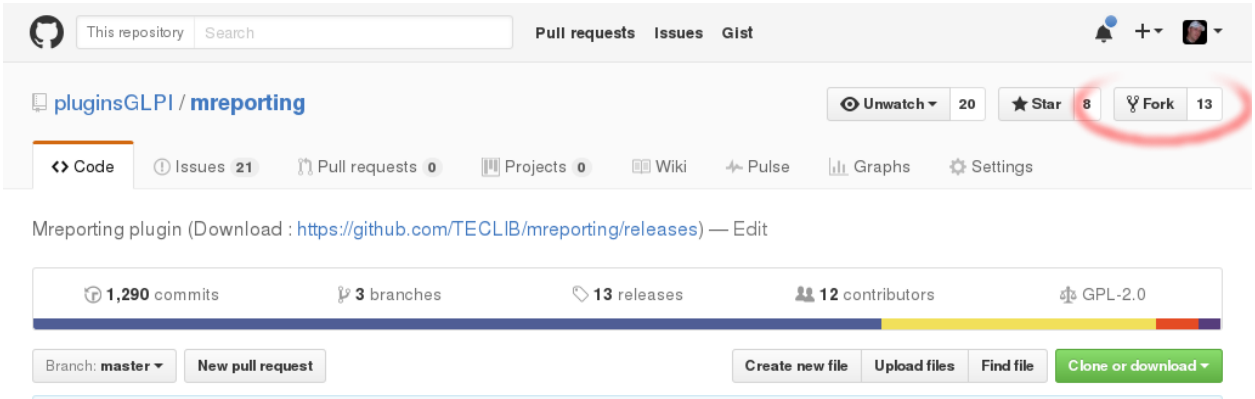


Fig. 2 – Le bouton fork

Depuis votre copie de travail, ajoutez un nouveau remote, que vous nommerez par exemple comme votre compte github (le nom n'a pas d'importance, il suffit que vous vous en souveniez, et que vous restiez cohérent dans les autres projets). En prenant soin de remplacer `{github_username}` avec votre propre nom d'utilisateur, lancez :

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

Toutes les branches que vous allez créer et qui doivent être revues seront publiées sur votre fork.

1.4 Initialisation

Initialiser git-flow est assez simple, clonez le dépôt, aller sur la branche `master` et lancez :

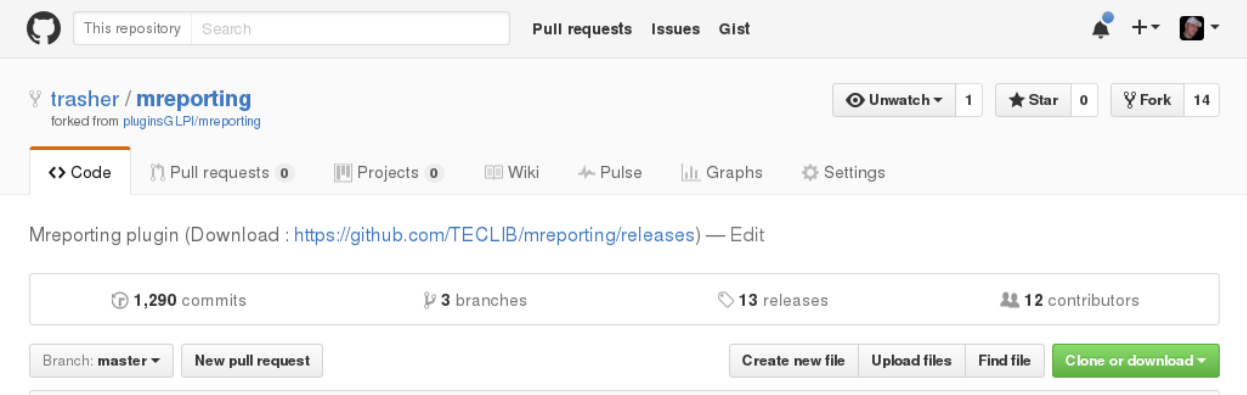


Fig. 3 – Le dépôt forké sur mon compte personnel

Note : Quand vous clonez un dépôt git, la branche par défaut sera utilisée. Dans la plupart des cas, ce sera `master`, mais pensez à vérifier.

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

Vous pouvez considérer que les réponses par défaut sont toutes correctes. Si la branche `develop` existe déjà, elle sera utilisée, le processus la créera sinon.

1.5 Processus non terminé

À certaines occasions, une commande git-flow peut ne pas se terminer (dans le cas d'un conflit par exemple). Ce n'est pas un problème, puisque c'est totalement géré :)

Si un processus git-flow est stoppé, corrigez simplement l'erreur et lancez la même commande une fois de plus. Il lancera tout simplement les tâches restantes.

Note : Pour vous assurez que tout a fonctionné correctement, regardez toujours attentivement la sortie !

1.6 merge vs rebase

Faut-il utiliser merge ou rebase ? Hé bien, c'est à vous de voir !

Avertissement : Bien que les deux solutions puissent être utilisées, et que vous avez la possibilité de choisir à chaque fois, n'oubliez pas qu'un `rebase` peut être destructeur ! Gardez cela à l'esprit.

En fait, vous pouvez toujours corriger un rebase depuis votre copie de travail locale (en utilisant `reflog`). Notez cependant que c'est quelque chose que vous ne devriez pas utiliser si vous n'êtes pas un expert git ;)

Je ne souhaite nourrir aucun troll ; les deux possèdent leurs avantages et leurs inconvénients. Mon conseil est d'éviter les commit de merge quand ils ne sont pas utiles. Je vais essayer d'expliquer quelques cas de figure standard, et la manière dont je les gère à travers les quelques exemples qui suivent...

Vous travaillez sur une *feature* ; tout a été concentré dans un seul et unique commit. Par défaut, le processus git-flow process ajoutera votre commit sur la branche `develop` et ajoutera un commit de merge (vide) également. Ce dernier n'est vraiment pas utile, il rend juste l'historique moins lisible. Si le commit de merge n'est pas vide, les choses commencent à se compliquer ; vous avez probablement loupé un `git flow feature rebase` quelque part.

Conclusion : utilisez **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 : utilisez **merge**

Vous venez de terminer une *feature*, tout comme quelqu'un d'autre… Mais l'autre a déjà poussé ses modifications sur la branche `develop` distante. Si vous lancez un `(develop) $ git push`, un message vous informera que vous ne pouvez pousser car la branche distante a changé.

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 : utilisez **rebase**.

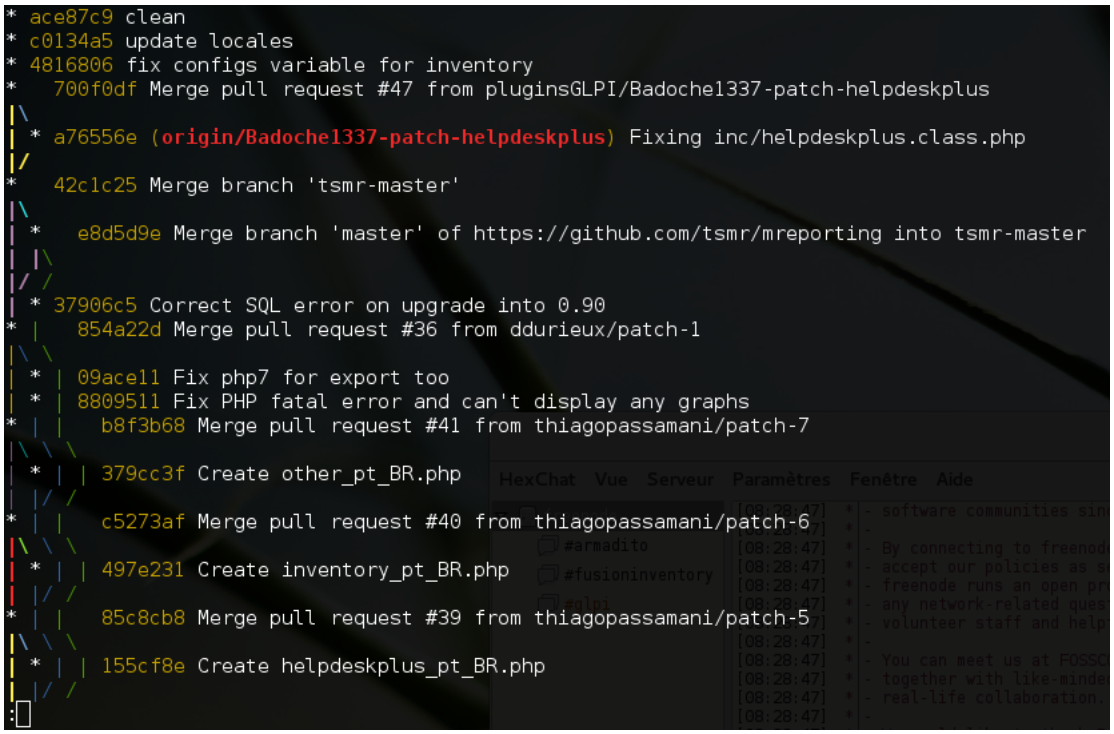


Fig. 4 – Un exemple d'historique (depuis le plugin mreporting).

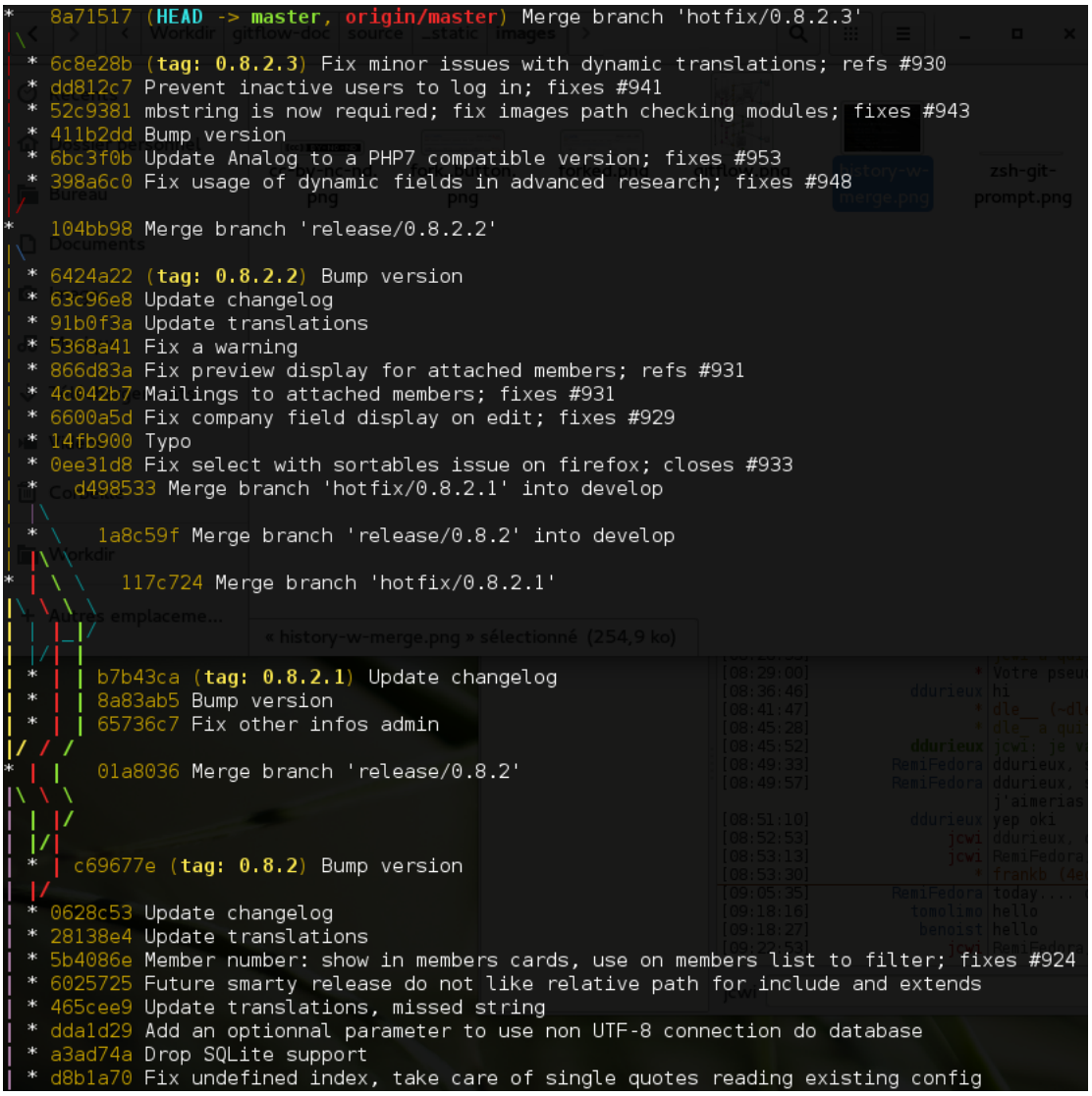


Fig. 5 – Un autre exemple d'historique (depuis le projet Galette).

CHAPITRE 2

Features

Avertissement : Prenez garde à ce que votre branche `develop` soit bien à jour avant de créer ou de terminer une feature ! Même si ça peut se corriger assez facilement ;)

Cette possibilité sera utilisée pour implémenter de nouvelles fonctionnalités dans le projet. Les features sont prévues pour être créées depuis la branche `develop` et pour être réintégrées dans la branche `develop` également.

Note : Vous pouvez avoir autant de feature que vous le souhaitez sur un projet ; mais vous vous apercevrez à l'usage que de travailler avec de trop nombreuses features peut rapidement devenir cauchemardesque, tout comme en maintenir trop longtemps ;)

2.1 Création

Le nom de la feature est à votre entière appréciation, choisissez quelque chose de simple et court, qui décrit ce que vous allez faire. Pour créer une feature nommée `my-great-feature`, vous entrerez :

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

Ce qui réalisera automatiquement les tâches suivantes :

1. créer une nouvelle branche `feature/my-great-feature` depuis la branche `develop`,
2. effecturea un checkout de la branche `feature/my-great-feature`.

Donc, oui, vous êtes parés à travailler ! Codez, commitez, poussez, ... Vous êtes sur une branche, vous pouvez faire ce que vous voulez (ou presque !).

Comme décrit dans *Travailler avec Github*, vous utiliserez votre fork lors de l'ajout de nouvelles branches. Vous y parviendrez en lançant :

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

2.2 Cycle de vie

Parfois, il ne s'est rien passé sur la branche `develop` avant que vous n'ayez terminé votre feature, vous n'aurez dans ce cas à vous soucier de rien.

Mais parfois, il est possible que d'autres features aient été ajoutées, ou que des bogues aient été corrigés… Vous devrez donc maintenir votre branche feature à jour. En considérant que votre branche `develop` soit à jour (mais vous maintenez toujours votre branche `develop` à jour, n'est-ce pas ? :p) :

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

Cela va rebaser votre branche feature par dessus le develop ; comme si vous veniez juste de la créer. Vos commits seront appliqués un à un par dessus. Expliquer le fonctionnement du rebase est hors de la portée de la présente documentation, mais vous pourrez trouver de nombreuses ressources là-dessus.

2.3 Pull Request

Votre feature est terminée, elle doit maintenant être revue avant de pouvoir être mergée. Poussez les modifications sur votre fork, allez sur sa page, sélectionnez votre branche et cliquez sur le bouton « New pull request ».

Vous pouvez fournir des détails complémentaires au besoin, soumettre, et enfin attendre qu'un autre développeur effectue la revue de vos changements ! Une fois acceptée, retournez sur votre copie de travail, et suivez les instructions du paragraphe ci-dessous.

2.4 Finalisation

Une fois terminé, et votre PR acceptée, vous pourrez nettoyer un peu l'historique de votre branche, regrouper vos commit pour éviter de conserver des messages de commit du type « Oups, j'ai oublié… ». En admettant que votre copie de travail soit sur la branche de votre feature, lancez :

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

Ce qui réalisera les tâches suivantes :

1. merge de la branche `feature/my-great-feature` dans `develop`,
2. vous demandera un message de commit (qui sera par défaut « Merge branch "feature/my-great-feature" into develop »)
3. supprimera la branche `feature/my-great-feature`

Pour la seconde étape, vous pouvez juste laisser le message tel que ; si vous avez regroupé vos commits, vous pourrez le supprimer simplement en utilisant :

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

Ou pas, *à vous de voir* :)

Pour terminer, poussez la branche `develop` pour que le dépôt distant soit à jour ! Et c'est terminé, *my-great-feature* a été réintégrée dans `develop` et fera partie de la prochaine release ! Félicitations o/

N'oubliez pas de supprimer la branche `feature/my-great-feature` distante :

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

CHAPITRE 3

Hotfix

Avertissement : 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 !

Vous utilisez un `hotifx` pour corriger des bogues dans la dernière version stable du projet, peu importe qu'il s'agisse d'une version *majeure*, *mineure* ou *patch*.

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

3.1 Création

Le nom du hotfix doit être la version qu'il va devenir. Si la dernière version était *1.3.2* ; vous créerez un hotfix *1.3.3* en utilisant :

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

Ce qui réalisera automatiquement les tâches suivantes :

1. création d'une nouvelle branche nommée `hotfix/1.3.3` depuis la branche `master`,
2. récupération (`checkout`) de la branche `hotfix/1.3.3`

3.2 Cycle de vie

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 :

— renommer votre branche *hotfix*
— mettre le code à jour

En partant du principe que la version *1.3.3* a été publiée depuis un autre *hotfix*, vous travaillerez sur la version *1.3.4* :

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

3.3 Pull Request

Votre *hotfix* est terminé, il doit maintenant être revu avant de pouvoir être mergé. Poussez les modifications sur votre fork, allez sur sa page, sélectionnez votre branche et cliquez sur le bouton « new pull request ».

Vous pouvez fournir des détails complémentaires au besoin, soumettre, et enfin attendre qu'un autre développeur effectue la revue de vos changements ! Une fois acceptée, retournez sur votre copie de travail, et suivez les instructions du paragraphe ci-dessous.

3.4 Finalisation

Avertissement : Avant de lancer les commandes pour terminer votre *hotfix*, assurez-vous que :

— votre branche `master` soit à jour
— aucun autre *hotfix* utilisant la même version n'ait été mergé (utilisez `git tag | sort -V`)

Avertissement : Vous **devez utiliser Git en ligne de commande, et non les possibilités de Github** pour terminer le hotfix !

Terminer un *hotifx* est aussi simple que :

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

Cale va :
— Merger les changements dans la branche `master`,
— Créer un tag `1.3.4`,
— Merger les changements dans la branche `develop`,
— Supprimer votre branche `hotfix/1.3.4` locale.

Une fois que votre *hotfix* est terminé ; vous devrez pousser `master`, `develop` ainsi que les `tags`, et aussi supprimer la branche `hotfix/1.3.4` distante :

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

CHAPITRE 4

Releases

Avertissement : Prenez garde à ce que votre branche `develop` soit bien à jour avant de créer une *release*, et que vos branche `master` et `develop` soient toutes deux à jour avant de la terminer !

Vous aurez recours à la fonctionnalité *release* pour publier de nouvelles versions *mineures* ou *majeures*, mais pas pour les *patches*. Ceci est prévu pour publier de nouvelles versions depuis la branche `develop`.

Note : Vous pouvez avoir plusieurs *release* sur un même projet, mais en toute honnêteté, je ne parviens pas à trouver une cas de figure où ce serait réellement utile. À vous de voir ;)

4.1 Création

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
```

Ce qui réalisera automatiquement les tâches suivantes :

1. création d'une nouvelle branche nommée `release/1.4.0` depuis la branche `develop`,
2. récupération (`checkout`) de la branche `release/1.4.0`

4.2 Cycle de vie

Avertissement : Jusque ce soit terminé, vous pouvez toujours ajouter de nouveaux *hotfix* ou *features* (dans tous les cas, si ne nouvelle *feature* doit être ajoutée à votre *release*, vous avez probablement un souci de planning ;)).

Mais gardez à l'esprit que **rien ne sera ajouté à votre branche release sans que vous l'ayez fait vous-même !**

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.

Ce type de branche peut être utilisé pour tester également.

Parfois, vous pourrez également créer une *release* pour la terminer immédiatement sans effectuer de modifications. . . :-)

Si un nouveau *hotifx* a été ajouté, vous devrez l'incorporer à votre branche *release*. Pour savoir comment procéder, vous devrez tout d'abord déterminer si quelque chose d'autre a changé, vous ne souhaiterez probablement pas qu'une *feature* terminée après que vous ayez décidé de publier ne soit récupérée.

Note : Souvenez-vous qu'il est toujours préférable de merge et de faire un cherry-pick plutôt que de reporter des modifications manuellement ; ceci pour éviter des conflits ultérieurs lors de la finalisation.

Dans le cas le plus simple, rien n'a changé dans le `develop`, mettez juste à jour et lancez :

```
(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.

Si vous vous trouvez dans le second cas de figure, poussez le derniers changements sur votre fork, allez sur votre page github, sélectionnez votre branche et cliquez sur le bouton « New pull request ».

Vous pouvez fournir des détails complémentaires au besoin, soumettre, et enfin attendre qu'un autre développeur effectue la revue de vos changements !

Une fois acceptée, ou si vous n'avez pas besoin de passer par une pull request, retournez sur votre copie de travail, et suivez les instructions du paragraphe ci-dessous.

4.4 Finalisation

Avertissement : Avant de lancer les commandes pour terminer votre *release*, assurez-vous que :

- vos branches `master` et `develop` soient à jour
- aucun autre tag utilisant la même version n'ait été créé (utilisez `git tag | sort -V`)

Avertissement : Vous **devez utiliser Git en ligne de commande, et non les possibilités de Github** pour terminer la release !

Terminer une *release* est aussi simple que :

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

Cela va :

- Merger les changements dans la branche `master`,
- Créer un tag `1.4.0`,
- Merger les changements dans la branche `develop`,
- Supprimer votre branche `release/1.4.0` locale.

Une fois que votre *release* est terminée, vous devrez pousser `master`, `develop` ainsi que les `tags` ; et aussi supprimer la branche distante `release/1.4.0` (si toutefois elle existe) :

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

CHAPITRE 5

Références

Quelques références et liens sur git-flow.

- explication du modèle (auteur original)
- *extension git <https ://github.com/nvie/gitflow>* (auteur original)
- un petit tutoriel
- un autre tutoriel, par Atlassian
- un cheat sheet
- un tutoriel vidéo en français
- questions stack overflow

Quelques liens à propos de Git lui-même :

- site web officiel
- le livre ProGit visible en ligne ou téléchargeable ; il a également été imprimé
- tutoriels Atlassian
- un cheat sheet interactif que je trouve particulièrement intéressant pour comprendre les interactions et les différents niveaux
- une publication Stanford avec plusieurs traductions

5.1 Outils

J'utilise vim pour coder, et l'extension git pour gérer mon git-flow ; et bien que des plugins spécifiques existent, je ne les utilise pas personnellement.

Dans tous les cas, si vous utilisez un outil qui implémente git-flow nativement ou via un plugin, n'hésitez pas à ouvrir une PR sur cette documentation !

(cc) BY-NC-ND