Gestion de version avec git

Table des matières

Avant-propos . . . 2
La référence absolue (selon moi) . . . 2
1. Environnement . . . 2
2. Principes généraux . . . 3
2.1. Git c'est quoi ? . . . 4
2.2. Concepts clefs . . . 4
2.3. Avant de commencer . . . 4
3. Scénario **classique** (et idéal) . . . 5
3.1. Etape 1 : création du *repository* local . . . 5
3.2. Etape 2 : ajout des fichiers . . . 6
3.3. Etape 2 (suite) : vérification . . . 6
3.4. Etape 3 : Commit . . . 6
3.5. Etape 3 (suite) : Gestion "locale" . . . 7
3.6. Etape 4 : Trouver un hébergement distant . . . 7
3.7. Etape 4 (suite) : déclarer le dépôt distant . . . 7
3.8. Etape 5 : branch, edit, commit, merge . . . 8
3.9. Etape 5 (suite) : branching . . . 8
3.10. Etape 5 (suite) : edit . . . 8
3.11. Etape 5 (suite) : commit . . . 9
3.12. Etape 5 (suite) : utilisation des branches . . . 9
3.13. Etape 5 (suite) : merge . . . 9
3.14. Etape 6 : push . . . 10
3.15. Etape 7 : pull request (demande) . . . 10
3.16. Etape 7 (suite) : pull request (acceptation) . . . 10
3.17. Dépôts existants . . . 11
4. Principes de bases . . . 11
4.1. Les 3 espaces . . . 11
4.2. Stash . . . 11
5. Illustration des branches . . . 12
6. Bonne utilisation . . . 16
6.1. Avoir une procédure concertée . . . 16
6.2. Ne pas versionner n'importe quoi! . . . 17
6.3. Les "releases" . . . 17
7. La gestion de version n'est pas un long fleuve tranquille . . . 18
7.1. Oups! j'ai oublié un truc . . . 18
7.2. Oups! j'ai mis trop de truc . . . 18

7.3. CTRL+Z . . . 18
7.4. Où j'en suis . . . 19
8. Gestion des branches . . . 20
9. Les différents **merge** . . . 21
9.1. Explicit `merge` . . . 21
9.2. Implicit `merge` . . . 21
9.3. Implicit `merge` . . . 22
9.4. Squash on `merge` . . . 22
9.5. `merge` vs. `rebase` . . . 22
10. Gestion des conflits . . . 23
10.1. À la main . . . 24
10.2. Avec un peu d'aide . . . 24
11. Git avancé . . . 24
11.1. Les outils clonés de git dans un dépôt git . . . 24
11.2. Git-Flow . . . 25
11.3. Résumé des commandes . . . 25
11.4. Liens utiles . . . 25
11.5. Glossaire . . . 26

Avant-propos

La référence absolue (selon moi)

https://git-scm.com/

https://git-scm.com/book/fr/v2

1. Environnement

Vous pouvez installer `git` depuis le site officiel. Nous utilisons ici la version `2.32.1` en ligne de commande. Mais vous pouvez bien sûr utiliser un client graphique (comme https://www.sourcetreeapp.com ou https://www.gitkraken.com/).

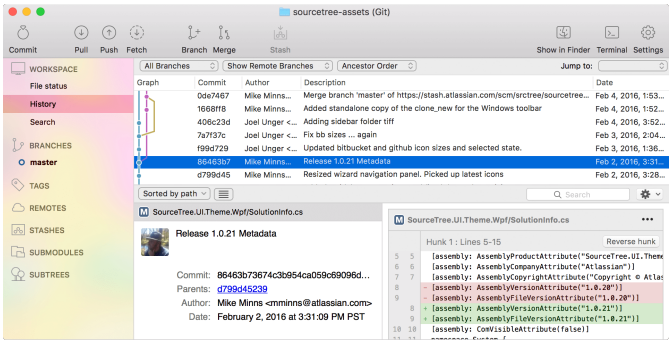


Figure 1. Client graphique (https://www.sourcetreeapp.com)

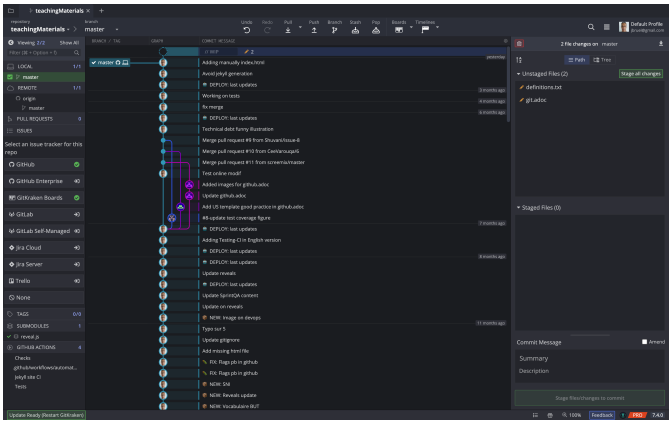


Figure 2. Client graphique (https://www.gitkraken.com/)

Figure 3. Lignes de commande

Je vous recommande tout de même de connaître les commandes en ligne, c'est souvent utile! Pour vous familiariser avec les commandes **git** en ligne, ne pas hésiter à utiliser le site https://learngitbranching.js.org/ (disponible en français si besoin).

2. Principes généraux

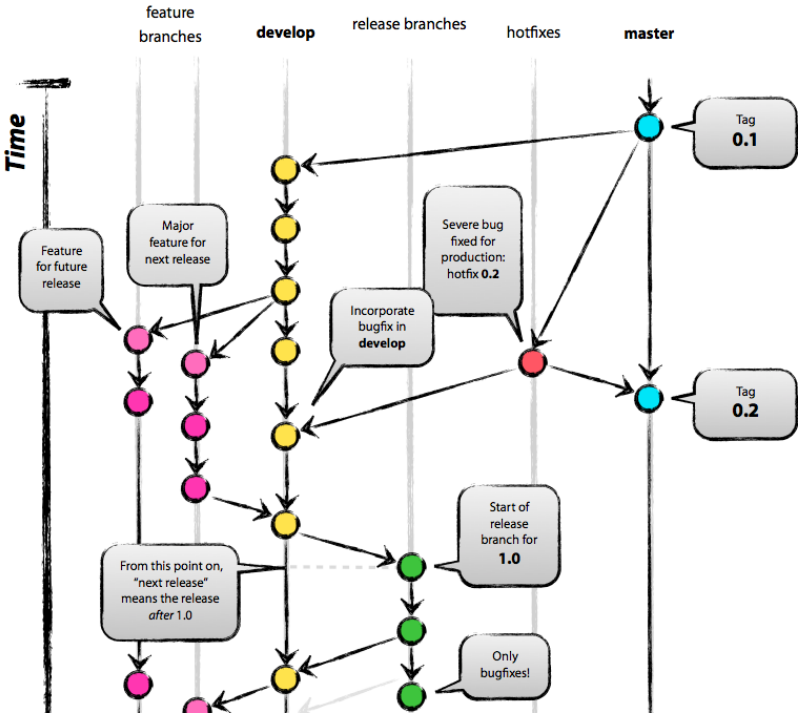
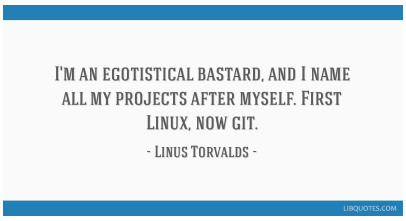


Figure 4. Usage classique de git (http://nvie.com/posts/a-successful-git-branching-model/)

Ce modèle est controversé et ne fait pas l'unanimité. Nous verrons dans la suite qu'il y a plus simple.

2.1. Git c'est quoi ?

- Un programme de gestion de version de code source (VCS — ***V**ersion **C**ontrol **S**ystem*)
- Gratuit et open source
- Distribué (contrairement à Subversion par exemple)
- Créé à Linus Torvalds (2005)



2.2. Concepts clefs

- Répertoire (*directory* or *folder*)
- CLI: ***C**ommon **L**ine **I**nterface*
- Dépôt (*repository*)
- `GitHub.com` (ou `GitLab.com` ou `gitlab.iut-blagnac.fr`)

2.3. Avant de commencer

On initialise certaine variables (une fois pour toute en général) :

```
$ git config --global user.name "JM Bruel"
$ git config --global user.email jbruel@gmail.com
$ git config --global alias.co checkout
```

Ces informations sont stockées dans le fichier `~/.gitcongif`.

Voici un extrait du mien :

```
[user]
        name = Jean-Michel Bruel
        email = jbruel@gmail.com
[alias]
        co = checkout
        st = status
        repo = config --get remote.origin.url
```

Ce qui donne :

```
$ git co
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
$ git checkout
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
$ git repo
https://github.com/jmbruel/teachingMaterials
```

3. Scénario classique (et idéal)

3.1. Etape 1 : création du *repository* local

On démarre la gestion de version :

```
$ git init
```

Génération d'un répertoire `.git` dans le répertoire courant.

```
$ git init
Initialized empty Git repository in /tmp/.git/

$ ll
total 0
drwxr-xr-x   3 bruel  admin   102 21 jul 17:29 ./
drwxr-xr-x  35 bruel  admin  1190 21 jul 17:29 ../
drwxr-xr-x  10 bruel  admin   340 21 jul 17:29 .git/
```

3.2. Etape 2 : ajout des fichiers

On ajoute les fichiers courants au dépôt :

```
$ git add .
```

- Ne pas forcément tout ajouter (`git add *.c` par exemple pour ne versionner que les sources).

- Pensez à créer un fichier `.gitignore` pour éviter d'ajouter les fichiers indésirables (comme les fichiers de `log`).

3.3. Etape 2 (suite) : vérification

On peut visualiser les actions en vérifiant l'**état courant** du dépôt :

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   Generalites.txt
#   deleted:    S3/128056_56.d
...
```

3.4. Etape 3 : Commit

Pour entériner les changements :

```
$ git commit -m "First draft"
[master (root-commit) 4f40f5d] First draft
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 titi.txt
 create mode 100644 toto.txt
```

- Retenez que le `commit` est uniquement local!
- Mais même en local, il est bien utile en cas de problème.

3.5. Etape 3 (suite) : Gestion "locale"

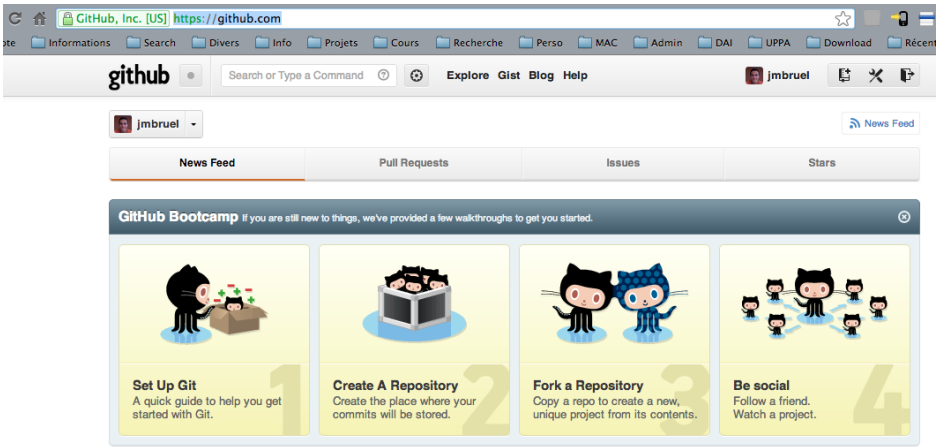
Exemple de scénario type (suppression exceptionnelle et rattrapage) :

```
$ rm titi.txt
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   deleted:    titi.txt
#
no changes added to commit (use "git add" and/or "git commit -a")

$ git checkout -f
$ ls titi.txt
titi.txt
```

3.6. Etape 4 : Trouver un hébergement distant

Il existe de nombreux endroits disponibles pour héberger du code libre. Les plus connus sont GitHub et GitLab.



3.7. Etape 4 (suite) : déclarer le dépôt distant

Après avoir créé un dépôt distant, il n'y a plus qu'à associer ce dépôt distant avec le notre.

```
$ git remote add origin git@github.com:jmbruel/first_app.git ①
$ git push -u origin master ②
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 225 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:jmbruel/first_app.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

① Il est possible d'avoir plusieurs dépôts distants, celui-ci sera référencé par `origin`.

② L'option `-u origin master` permet d'associer une fois pour toute les `git push` suivants au fait de "pousser" sur la branche `master` du dépôt `origin` (comme l'indique la dernière ligne).

3.8. Etape 5 : branch, edit, commit, merge

En cas d'édition et de commit local :

```
$ git checkout
Your branch is ahead of 'origin/master' by 1 commit.
```

3.9. Etape 5 (suite) : branching

git est très bon pour créer des branches :

```
$ git checkout -b testModifTiti
Switched to a new branch 'testModifTiti'
$ git branch
  master
* testModifTiti ①
```

① La branche courante est repérée par un `*`.

3.10. Etape 5 (suite) : edit

Après modification :

```
$ git status
# On branch testModifTiti
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   titi.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

3.11. Etape 5 (suite) : commit

On "sauvegarde" les changements :

```
$ git commit -am "modif de titi"
[testModifTiti 4515b5d] modif de titi
 1 files changed, 7 insertions(+), 0 deletions(-)
```

- On ne "sauvegarde" qu'en local!

3.12. Etape 5 (suite) : utilisation des branches

On peut "zapper" d'une branche à l'autre à volonté :

```
$ ll titi*
-rw-rw-r--    1 bruel  staff   331 12 nov 12:39 titi.txt

$ git co master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.

$ ll titi*
-rw-rw-r--    1 bruel  staff     0 12 nov 12:40 titi.txt
```

3.13. Etape 5 (suite) : merge

Maintenant que la branche a été développée (testée, etc.) on peut l'intégrer à la branche principale :

```
$ git co master
Switched to branch 'master'

$ git merge testModifTiti
Merge made by recursive.
 titi.txt |    7 +++++++
 1 files changed, 7 insertions(+), 0 deletions(-)
```

- On peut ensuite détruire la branche devenue inutile `git branch -d testModifTiti`.
- C'est une bonne habitude à prendre.
- Notez que l'historique des modifications (ainsi que les messages de commits successifs ne sont pas perdus).

3.14. Etape 6 : push

Maintenant que notre dépôt est satisfaisant, on peut le synchroniser avec le dépôt distant :

```
$ git push
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 977 bytes, done.
Total 9 (delta 2), reused 0 (delta 0)
To git@github.com:jmbruel/first_app.git
   6103463..3aae48a  master -> master
```

3.15. Etape 7 : pull request (demande)

3.16. Etape 7 (suite) : pull request (acceptation)

```
$ git checkout -b develop origin/develop
$ ...                          ①
$ git checkout master
$ git merge --no-ff develop ②
$ git push origin master    ③
```

① Vérifiez ce qui va être intégré

② On merge localement pour gérer les problèmes

③ On pousse sur master

3.17. Dépôts existants

Si vous devez partir d'un dépôt existant :

```
$ git clone git@github.com:jmbruel/first_app.git
```

- Pour obtenir le nom du dépôt distant : `git remote -v`.
- Vous avez aussi le nom du dépôt distant dans le fichier `.git/config`.

4. Principes de bases

4.1. Les 3 espaces

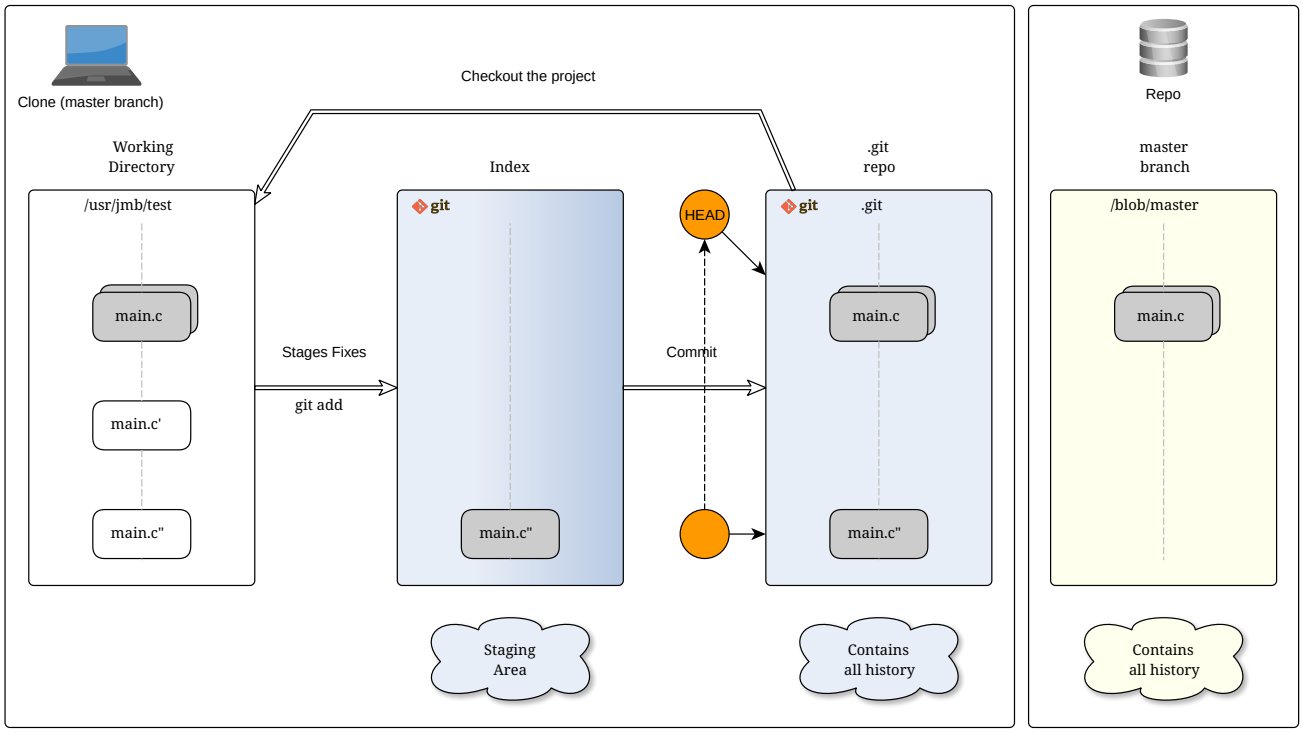


Figure 5. Les 3 espaces de git

4.2. Stash

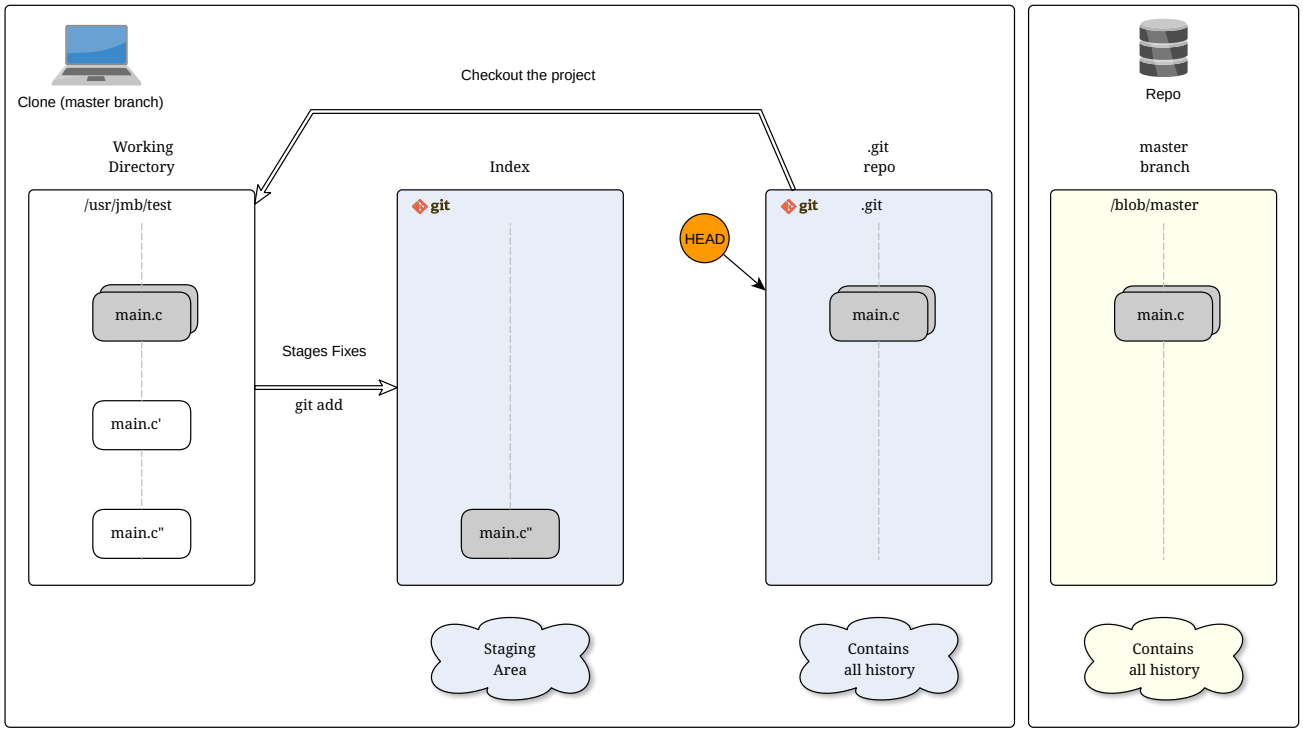


Figure 6. On a travaillé mais on ve revenir à une situation propre

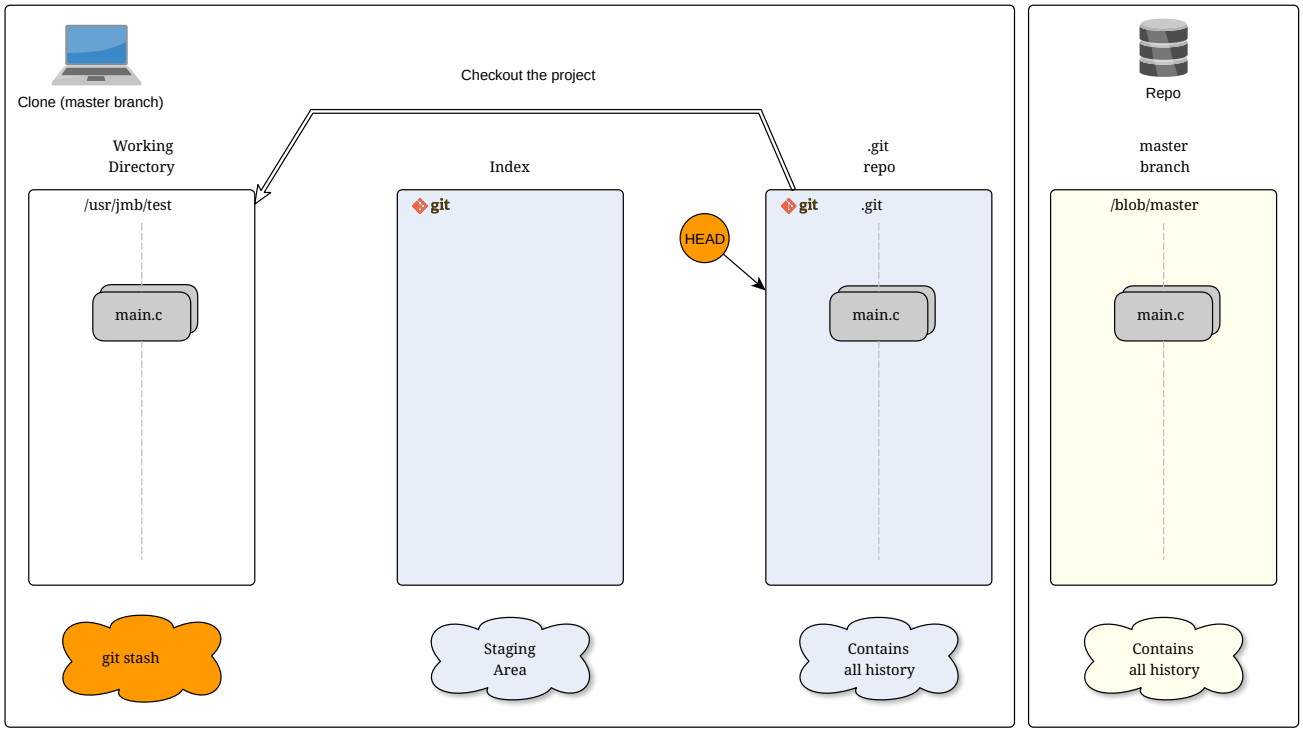


Figure 7. git stash

Même après un `git add` !

5. Illustration des branches

Voici une illustration de l'utilisation des branches (tirée de git-scm).

On part d'une situation type :

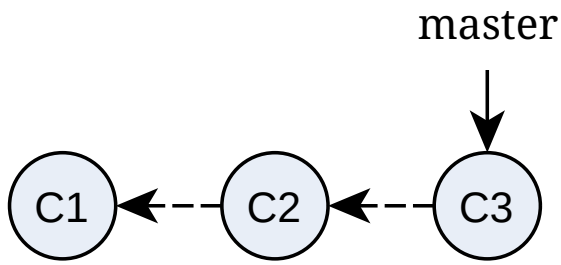


Figure 8. Situation initiale

On crée une branche (appelée `iss53` ici pour indiquer qu'elle traite de l'*issue* numéro 53) :

```
$ git checkout -b iss53
```

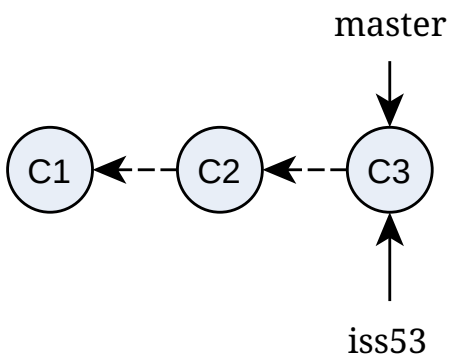


Figure 9. Création d'une branche

- git n'a créé qu'un pointeur ⇒ aucun espace mémoire perdu.

On modifie et on commit :

```
$ edit ...
$ git commit -m "blabla iss53"
```

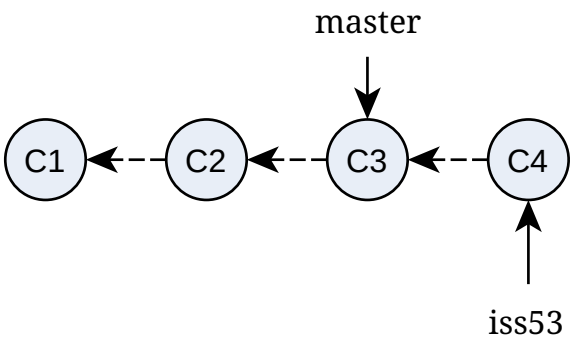


Figure 10. On commence à diverger de `master`

On revient à la branche maître pour tester une autre solution :

```
$ git checkout master
$ git checkout -b hotfix
$ edit ...
$ git commit -m "blabla hotfix"
```

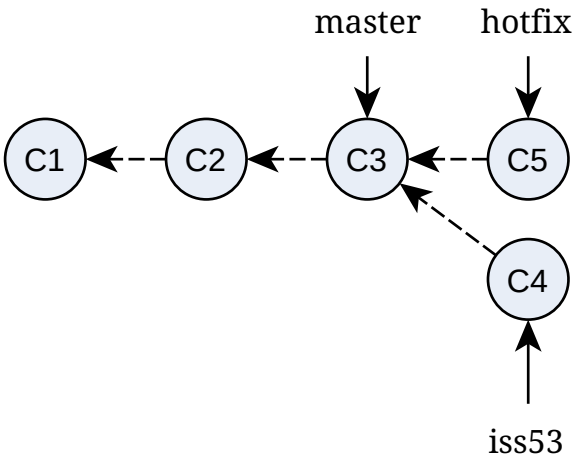


Figure 11. Maintenant on a 2 branches parallèles (en plus de `master`*)*

On intègre cette solution à la branche principale :

```
$ git checkout master
$ git merge hotfix
```

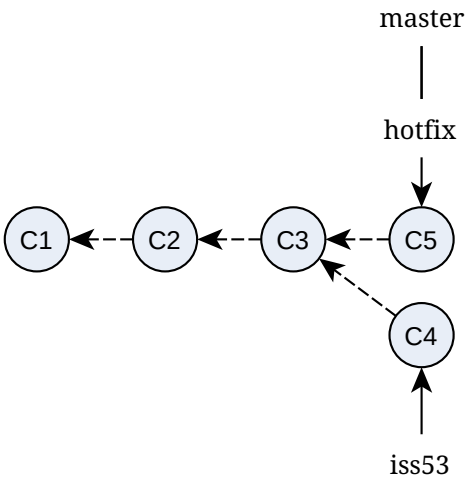


Figure 12. Merge de deux branches (en fast-forward)

- git utilise ici le **fast-forward**

On continue à travailler sur la branche `iss53` :

```
$ git branch -d hotfix ①
$ git checkout iss53
$ edit ...
$ git commit -m "blabla iss53"
```

① Destruction de la branche devenue redondante avec `master`.

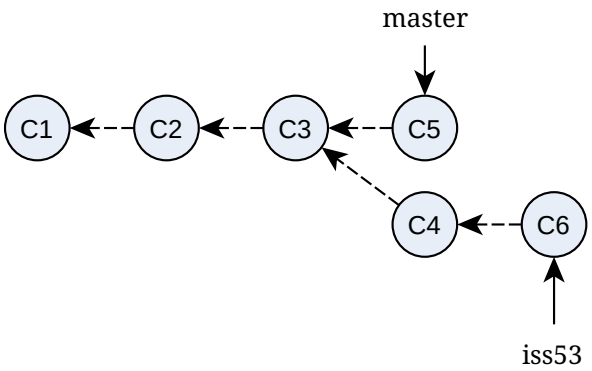


Figure 13. On retravaille sur `iss53`

On intègre cette branche :

```
$ git checkout master
$ git merge iss53
```

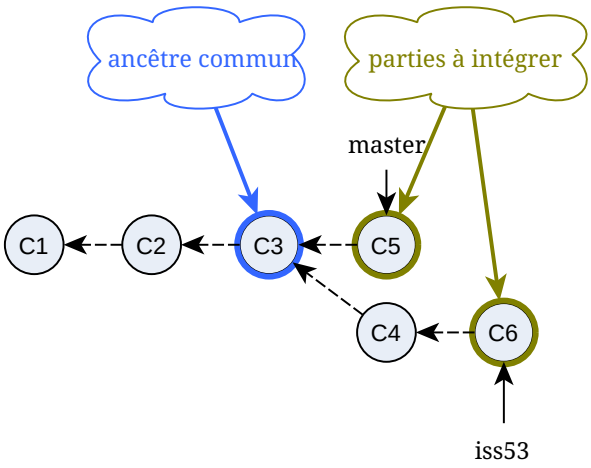


Figure 14. Explications du fonctionnement du merge sans fast-forward

- Explications : git recherche la racine commune (ici `c3` pour intégrer les branches (les commits feuilles) une par une et vérifier les conflits par itérations à partir de cette racine.

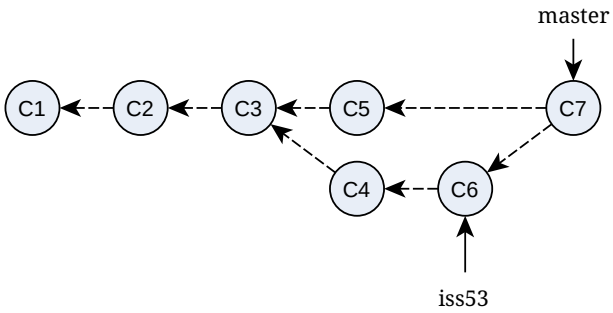


Figure 15. Situation finale

- On part du principe qu'il n'y a pas eu de Gestion des conflits
- On peut maintenant supprimer `iss53`

6. Bonne utilisation

6.1. Avoir une procédure concertée

Revenons sur l'exemple type :

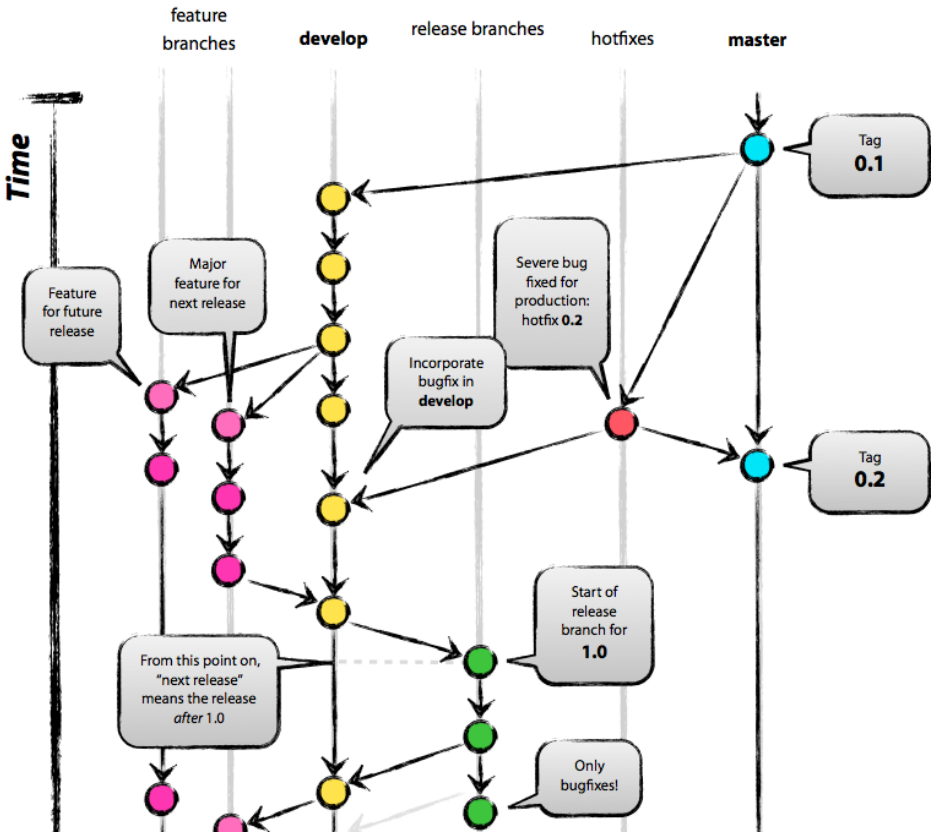


Figure 16. Usage classique de git (http://nvie.com/posts/a-successful-git-branching-model/)

6.2. Ne pas versionner n'importe quoi!

Ce qu'il ne faut pas versionner :

- les exécutables
- les zip dont le contenu change sans arrêt
- les images générées
- tous les binaires en général!

6.3. Les "releases"

En git on peut *taguer* des branches et c'est ce mécanisme qui permet de gérer simplement les *releases*. Dans l'exemple ci-dessous on tague le commit `ebb0a7` avec le tag `v1.0`.

```
$ git tag -a v1.0 ebb0a7 -m "Release 1.0 as required by client"
$ git tag
v1.0
$ git push origin v1.0
```

ne pas oublier de "pousser" le tag.

On peut voir les détails d'un commit tagué :

```
$ git show v1.0
tag v1.0
Tagger: Jean-Michel Bruel <jbruel@gmail.com>
Date:   Fri Sep 16 14:27:20 2016 +0200

Release 1.0 as required by client

commit 47da474098d95f8ef5c3ca838be8b87d7a7ed729
Author: Jean-Michel Bruel <jbruel@gmail.com>
Date:   Fri Sep 16 12:38:20 2016 +0200
```

On peut aussi taguer a posteriori :

```
$ git tag -a v1.2 9fceb02 ①
```

① ajoute le tag `v1.2` au commit dont le [SHA-1] commence par `9fceb02`

Par défaut les tags ne sont pas poussés sur le dépôt distant.

```
$ git push origin v1.5
```

7. La gestion de version n'est pas un long fleuve tranquille

7.1. Oups! j'ai oublié un truc

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

7.2. Oups! j'ai mis trop de truc

```
$ git add *.*
$ git reset *.class
```

Aucun danger

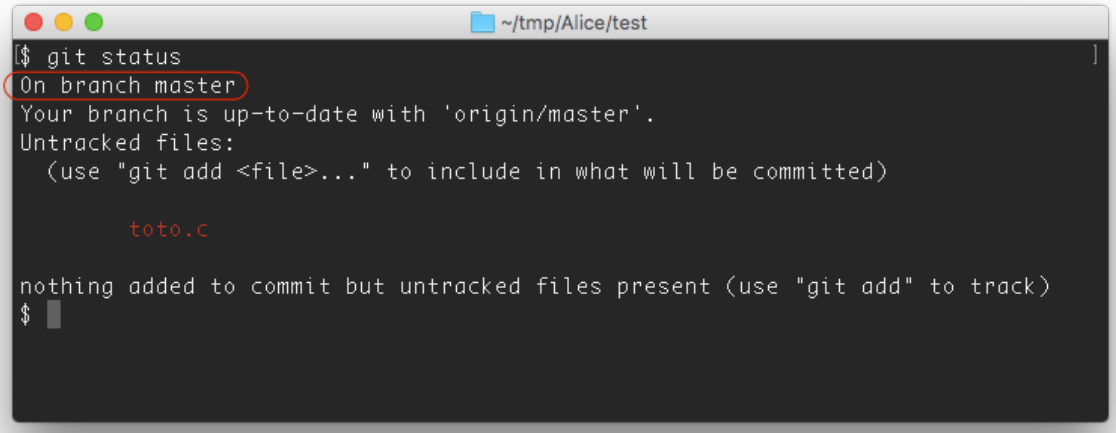
7.3. CTRL+Z

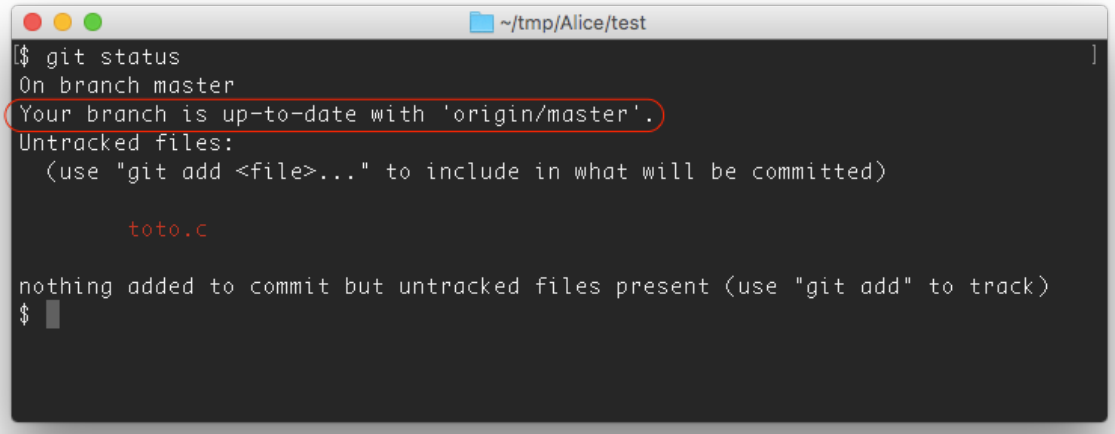
```
$ working on some file README.adoc ...
$ git checkout -- README.adoc
```

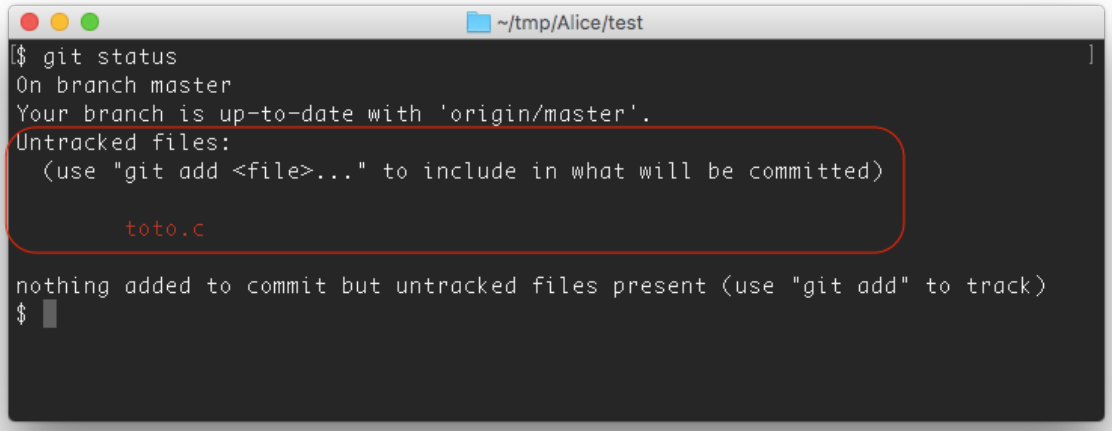
Danger!

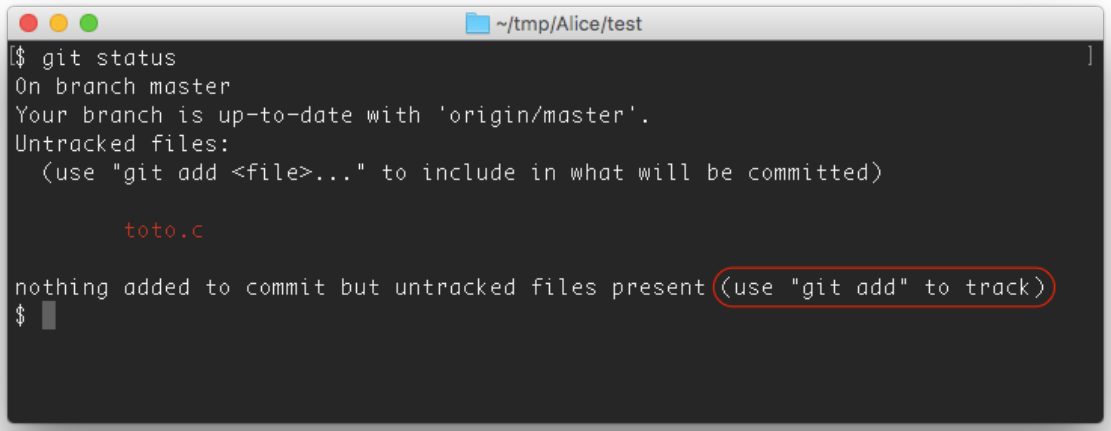
7.4. Où j'en suis

```
$ git status
```









8. Gestion des branches

La principale difficulté de git vient de la liberté en termes de branches.

Pour faire simple, je vous conseille une gestion qui marche bien pour les petites équipes, tiré de l'excellent livre Pro Git :

- Deux branches seulement: `master` et `develop`.

```
$ git branch
* develop     ①
  master      ②
```

① `develop` est la branche de travail qui contient la dernière version des codages en cours.

② `master` est toujours stable et sert au déploiement

- On fork `develop` pour traiter un *bug* ou une *feature*.

    - On merge dans `develop`
    - On détruit la branche devenue inutile

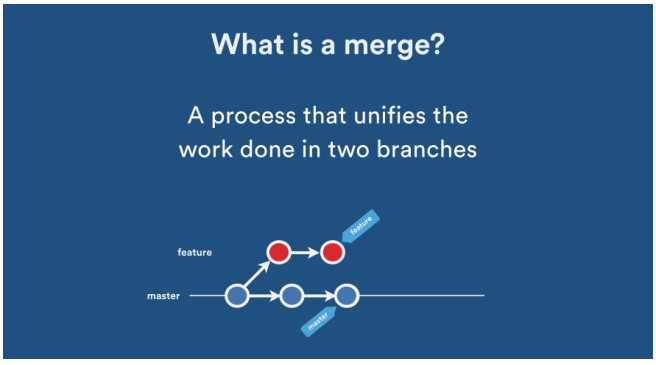
Ce qui donne le flot suivant dès que vous devez faire une amélioration (corriger un bug ou ajouter une fonctionnalités) :

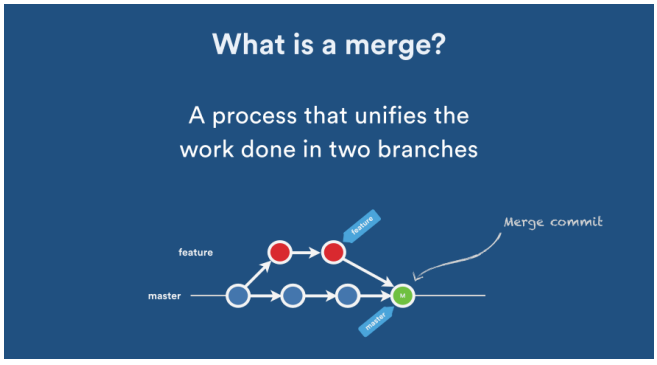
- Créer une branche (e.g., `fix-451`)
- Travailler sur cette branche
- Merger cette branche dans `develop`
- Rejouer les tests
- Régler les conflits éventuels
- Quand tout fonctionne ⇒ Etape 7 : pull request (demande)
- On peut livrer à partir de `master`

9. Les différents merge

Les illustrations suivantes (qui sont des GIF animés dans la version Web) sont tirés de : https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/?utm_source=twitter&utm_medium=social&utm_campaign=atlassian_pull-request-merge-strategies-the-great-debate.

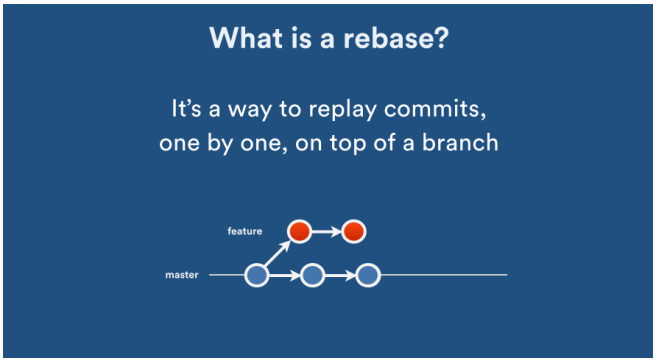
9.1. Explicit `merge`

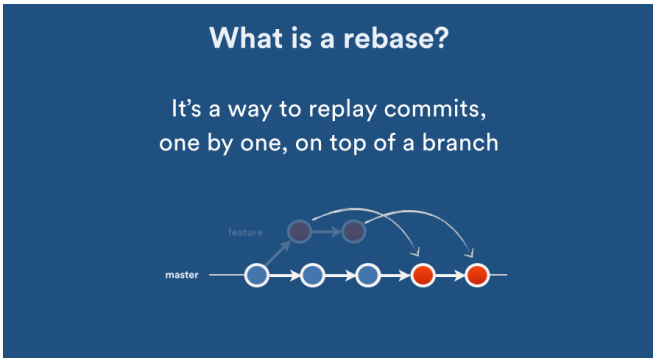




9.2. Implicit `merge`

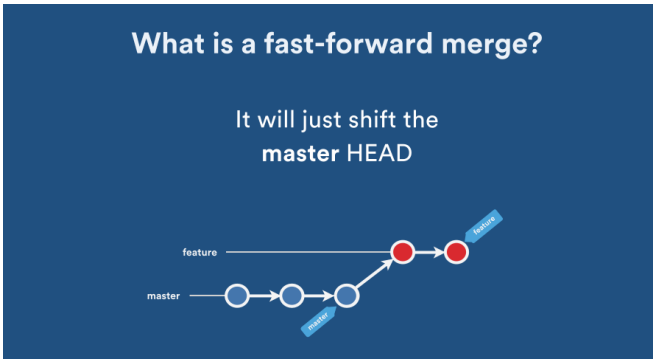
Via `rebase`

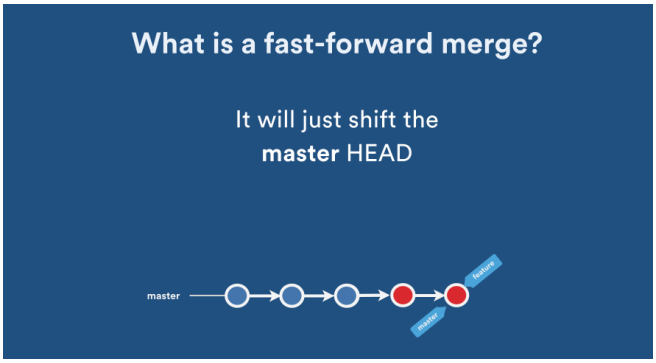




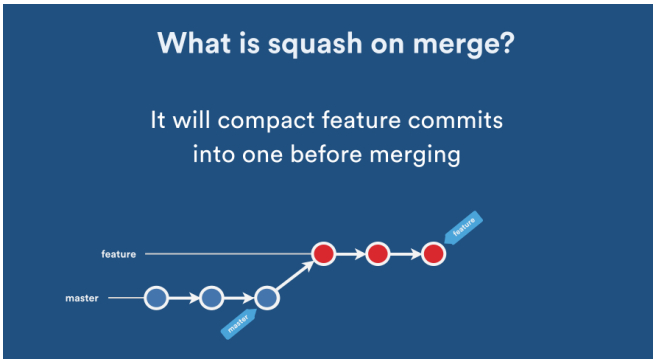
9.3. Implicit `merge`

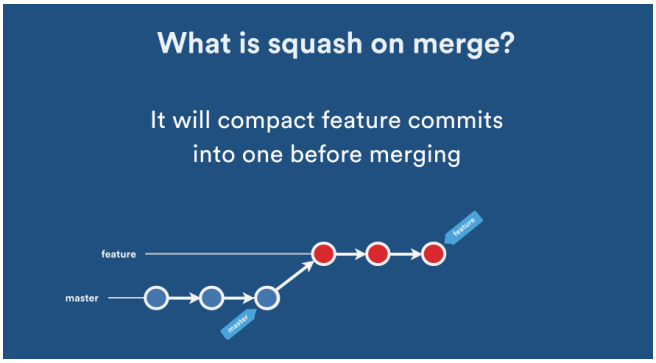
Via `fast-forward`





9.4. Squash on `merge`

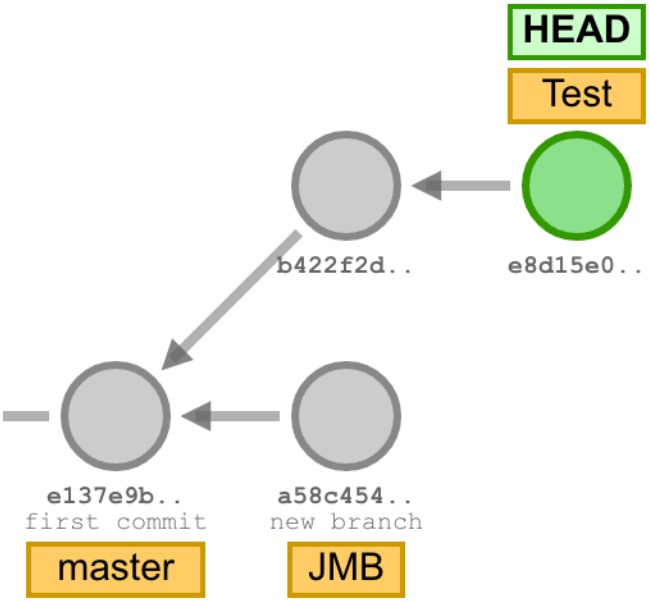




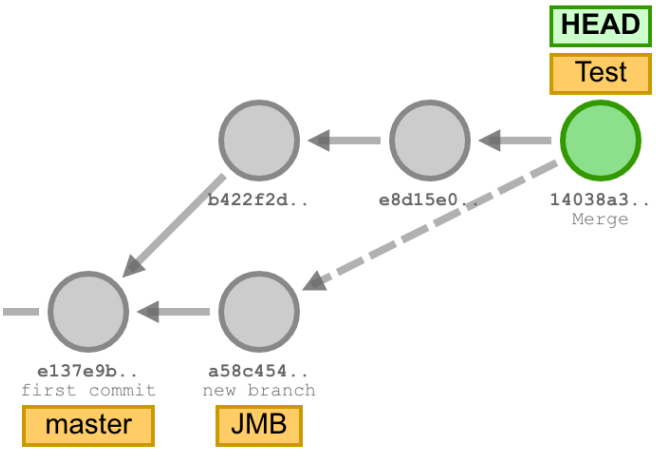
9.5. `merge` vs. `rebase`

Here is an illustration using http://git-school.github.io/visualizing-git/ :

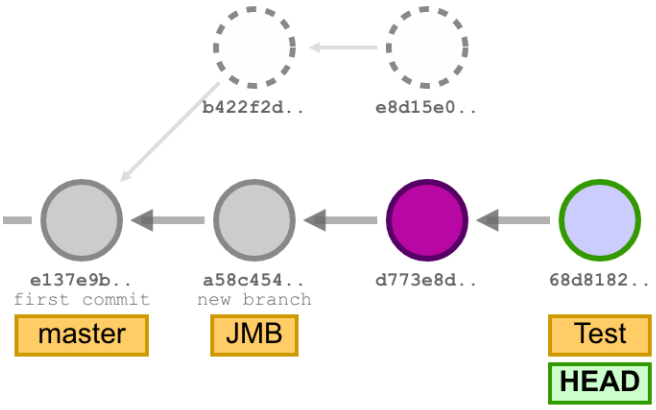
Initial situation:



`git merge JMB`:



`git rebase JMB`:



10. Gestion des conflits

La principale activité du programmeur qui utilise git en équipe vient de la gestion des **conflits**.

10.1. À la main

```
$ git checkout master
$ git merge other_branch
Auto-merging toto.txt
CONFLICT (content): Merge conflict in toto.txt
Automatic merge failed; fix conflicts and then commit the result.
$ more toto.txt
<<<<<<< HEAD ①
Salut monde
======= ②
hello world!

>>>>>>> other_branch ③
$ vi toto.txt ④
$ git commit ⑤
```

① Voilà où commence la différence entre la branche courante (`HEAD`) et la branche qu'on essaye de merger (`other_branch`)

② Séparation

③ Voilà où se termine cette différence

④ on édite le fichier à la main pour choisir la bonne version

⑤ on commit pour valider la modif

Il est déconseillé d'en profiter pour faire une nouvelle modif dans le fichier…

10.2. Avec un peu d'aide

- `git diff`
- `git difftool`
  - DiffMerge
  - …

11. Git avancé

11.1. Les outils clonés de git dans un dépôt git

Ne pas simplement cloner, car soucis de synchro plus tard.

Faire :

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Vous devez voir apparaître, en plus du répertoire cloné, un fichier `.gitmodules` (si c'est la 1ère fois).

11.2. Git-Flow

http://danielkummer.github.io/git-flow-cheatsheet/

11.3. Résumé des commandes

Voici un schéma pour résumer la philosophie (tiré de http://osteele.com) :

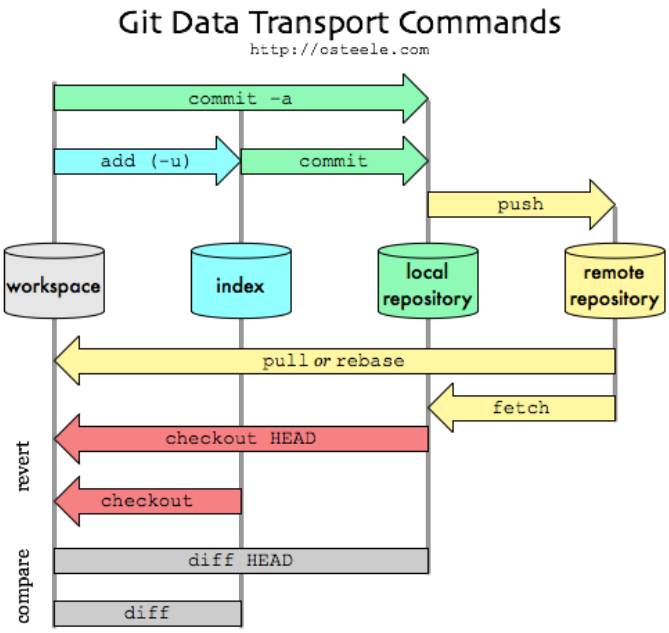


Figure 17. Résumé des commandes git (http://osteele.com)

11.4. Liens utiles

Le site de référence

http://git-scm.com/

Les "hébergeurs"

- GitHub
  - le site officiel
  - l'étiquette

- gitlab
  - le site officiel

Un excellent livre en ligne sur git

http://git-scm.com/book

Comparaison entre merge/rebase/etc.

https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/?utm_source=twitter&utm_medium=social&utm_campaign=atlassian_pull-request-merge-strategies-the-great-debate Un excellent tutoriel en Français et dynamique : http://learngitbranching.js.org/

Git pour les nuls

http://rogerdudler.github.io/git-guide/

Best practices

https://dev.to/bholmesdev/git-github-best-practices-for-teams-opinionated-28h7

11.5. Glossaire

fast_forward

Quand on merge une branche depuis un noeud situé sur le même "historique". Il s'agit donc pour git d'un simple déplacement de pointeur! Cf. illustration ici.

SHA-1

https://fr.wikipedia.org/wiki/SHA-1