

Gestion de version avec git

Table des matières

- [Avant-propos](#)
- [Pour suivre en live... 📺 📖 🖥](#)
- [Environnement](#)
- [!](#)
- [Principes généraux](#)
- [Avant de commencer](#)
- [-](#)
- [Scénario classique \(et idéal\).](#)
- - [Etape 1 : création du *repository* local](#)
 - [Etape 2 : ajout des fichiers](#)
 - [Etape 2 \(suite\) : vérification](#)
 - [Etape 3 : Commit](#)
 - [Etape 3 \(suite\) : Gestion "locale"](#)
 - [Etape 4 : Trouver un hébergement distant](#)
 - [Etape 4 \(suite\) : déclarer le dépôt distant](#)
 - [Etape 5 : branch, edit, commit, merge](#)
 - [Etape 5 \(suite\) : branching](#)
 - [Etape 5 \(suite\) : edit](#)
 - [Etape 5 \(suite\) : commit](#)
 - [Etape 5 \(suite\) : utilisation des branches](#)
 - [Etape 5 \(suite\) : merge](#)
 - [-](#)
 - [Etape 6 : push](#)
 - [Etape 7 : pull request \(demande\)](#)
 - [Etape 7 \(suite\) : pull request \(acceptation\)](#)
 - [Dépôts existants](#)
- [Illustration des branches](#)
- - [Illustration des branches \(suite\)](#)
 - [Illustration des branches \(suite\)](#)
 - [Illustration des branches \(suite\)](#)
 - [Illustration des branches \(suite\)](#)
 - [Illustration des branches \(suite\)](#)
 - [Illustration des branches \(suite\)](#)
 - [Illustration des branches \(suite\)](#)
- [Bonne utilisation](#)
- - [Avoir une procédure concertée](#)
 - [Ne pas versionner n'importe quoi!](#)
 - [Les "releases"](#)
 - [-](#)
 - [-](#)
 - [La gestion de version n'est pas un long fleuve tranquille](#)
 - [Oups! j'ai oublié un truc](#)
 - [Oups! j'ai mis trop de truc](#)
 - [CTRL+Z](#)
 - [Où j'en suis](#)

- -
- -
- -
- -
- [Gestion des branches](#)
- ◦ -
- -
- -
- -
- [Les différents merge](#)
- ◦ -
- [Explicit merge](#)
- [Implicit merge via rebase Or fast-forward](#)
- [Implicit merge via fast-forward](#)
- [Squash on merge](#)
- [merge VS. rebase](#)
- -
- -
- [Gestion des conflits](#)
- ◦ -
- [À la main](#)
- [Avec un peu d'aide](#)
- [Git avancé : Git-Flow](#)
- ◦ -
- [Wrap-up](#)
- ◦ [Résumé des commandes](#)
- [The End](#)

Avant-propos

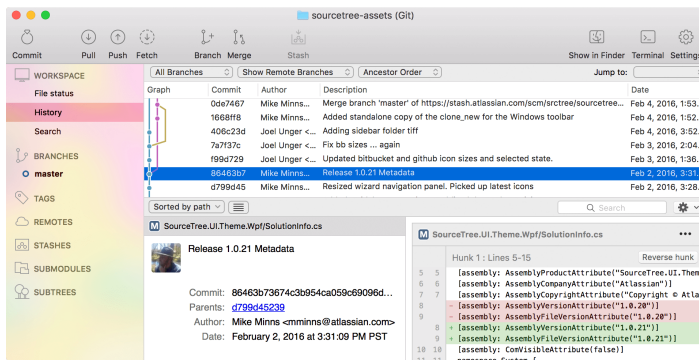
Pour suivre en live...   

<http://bit.ly/jmb-git>

<http://jmbhome.github.io/teachingMaterials/git.html>

Environnement

- [undefined](#) (V.2.24.3)
- <http://git-scm.com/>

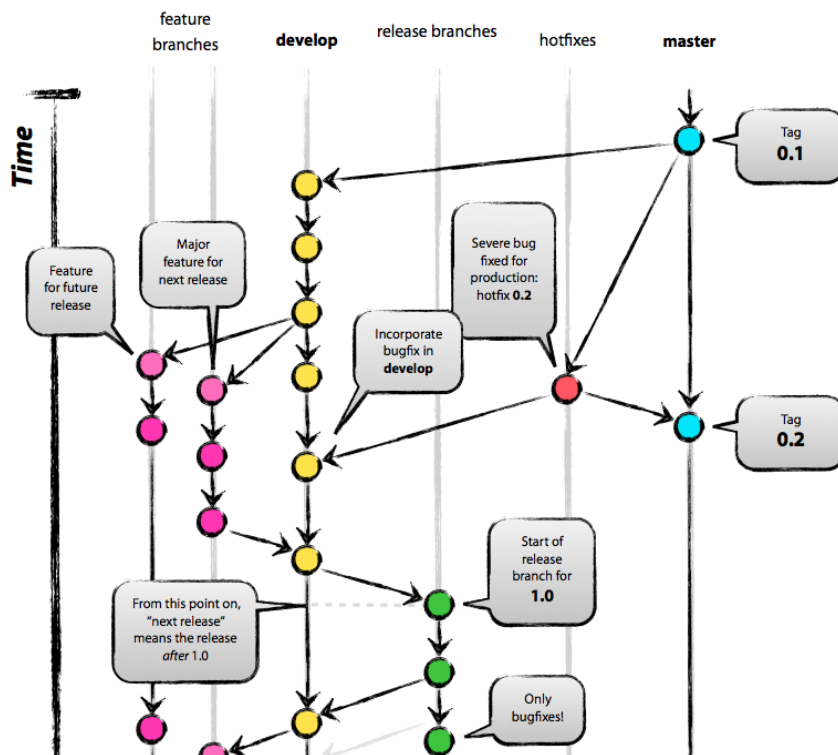


<https://www.sourcetreeapp.com>

!

```
MacBook-Pro-de-Jean-Michel:~ bruel$ git --version
git version 2.9.2
MacBook-Pro-de-Jean-Michel:~ bruel$
```

Principes généraux



<http://nvie.com/posts/a-successful-git-branching-model/>

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 ~/.gitconfig.

Voici un extrait [du mien](#) :

```
[user]
    name = Jean-Michel Bruel
    email = jbruel@gmail.com
[alias]
    co = checkout
    st = status
```

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)
```

Scénario classique (et idéal)

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/
```

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`).

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
...
```

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.

Etape 3 (suite) : Gestion "locale"

Exemple de scénario type (suppression exceptionnelle et rattrapage) :

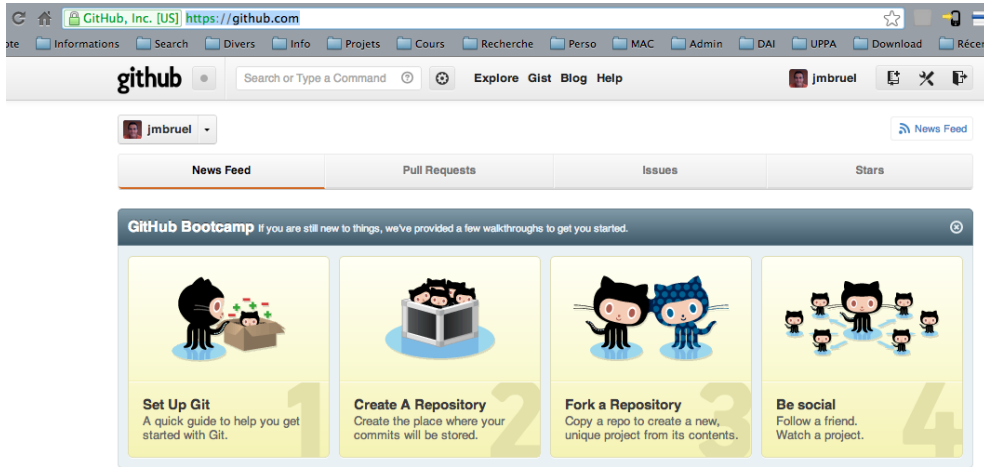
```
$ rm titi.txt
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    titi.txt
#
no changes added to commit (use "git add"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

$ git checkout -f
```

```
$ ls titi.txt
titi.txt
```

Etape 4 : Trouver un hébergement distant

Il existe de nombreux endroits disponibles pour héberger du code libre. Les plus connus sont [GitHub](#) et [GitLab](#).



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
$ git push -u origin master
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`.

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

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.
```

Etape 5 (suite) : branching

[undefined](#) est très bon pour créer des branches :

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

1 La branche courante est repérée par un *.

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")
```

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!

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
```

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).

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
```

Etape 7 : pull request (demande)

Etape 7 (suite) : pull request (acceptation)

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

- 1 Vérifiez ce qui va être intégré
- 2 On merge localement pour gérer les problèmes
- 3 On pousse sur master

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`.

Illustration des branches

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

On part d'une situation type :

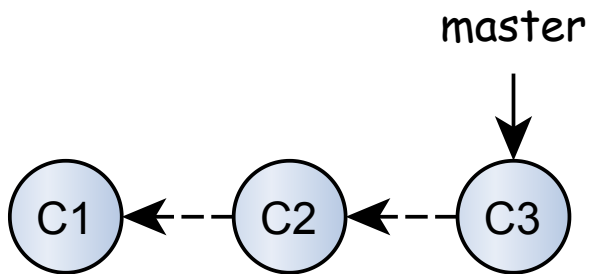
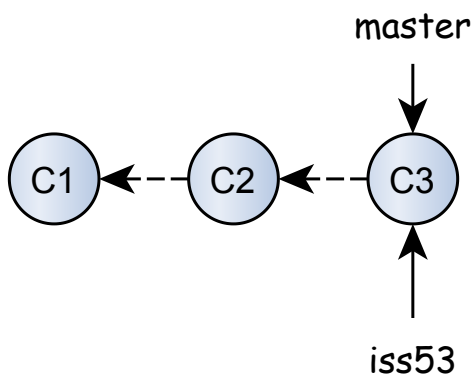


Illustration des branches (suite)

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

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

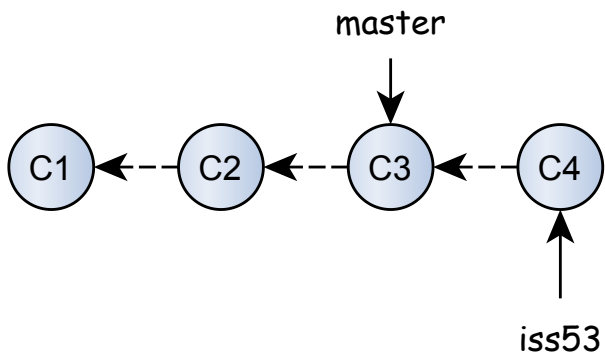


- [undefined](#) n'a créé qu'un pointeur \Rightarrow aucun espace mémoire perdu.

Illustration des branches (suite)

On modifie et on commit :

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



i On commence à diverger de `master`

Illustration des branches (suite)

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

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

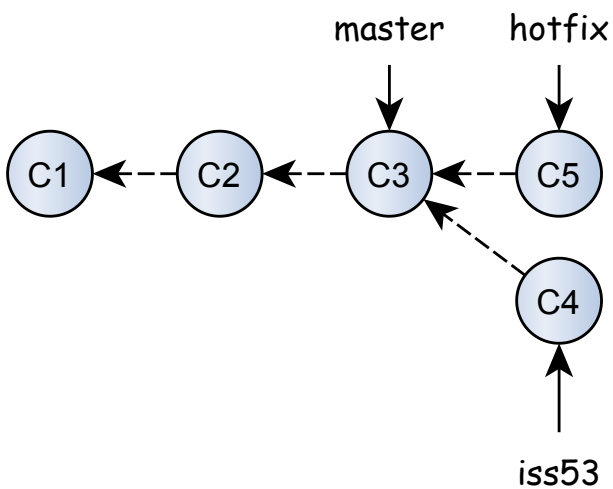
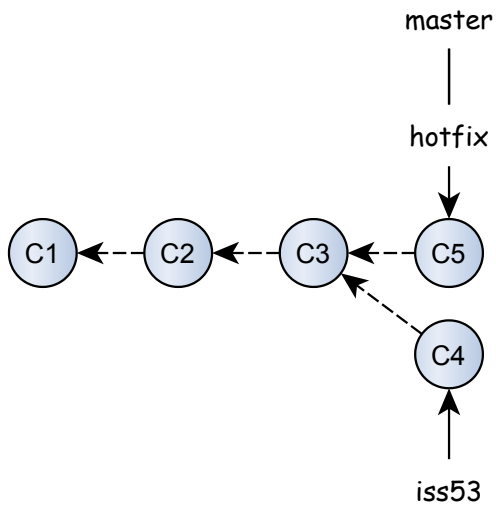


Illustration des branches (suite)

On intègre cette solution à la branche principale :

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



i Il manque le pointeur HEAD sur mes illustrations

i • [undefined](#) utilise ici le fast-forward

Illustration des branches (suite)

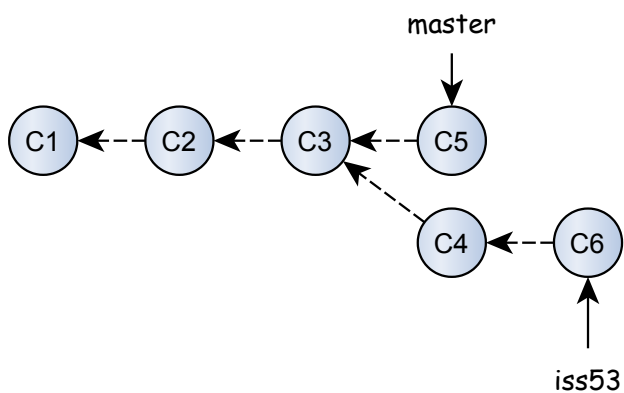
On continue à travailler sur la branche `iss53` :

```

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

```

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

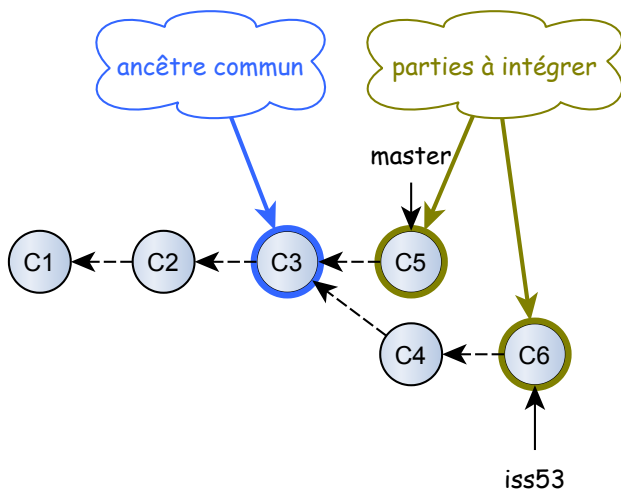


i On retravaille sur `iss53`

Illustration des branches (suite)

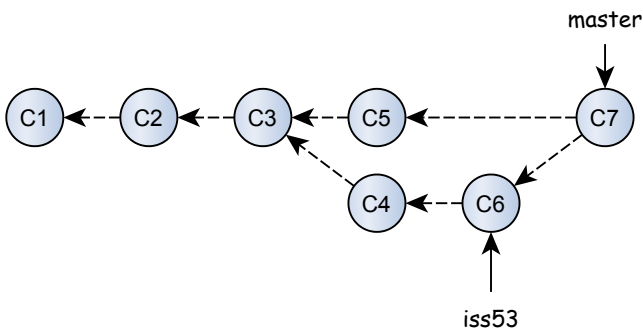
On intègre cette branche :

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



i Merge sans *fast-forward*

Illustration des branches (suite)



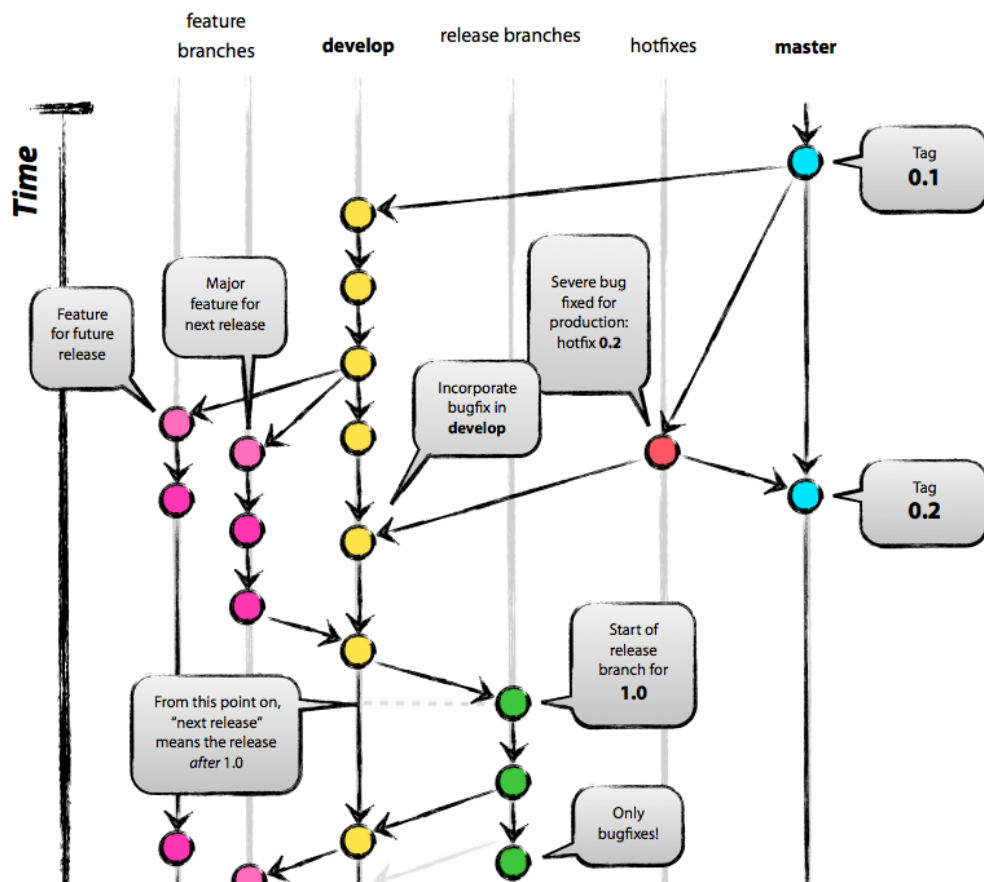
i Situation finale

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

Bonne utilisation

Avoir une procédure concertée

Revenons sur l'exemple type :



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!

Les "releases"

En [undefined](#) 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 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
```

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

Oups! j'ai oublié un truc

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

Oups! j'ai mis trop de truc

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



Aucun danger

CTRL+Z

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



Danger!

Où j'en suis

```
$ git status
```

```
~/tmp/Alice/test
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    toto.c

nothing added to commit but untracked files present (use "git add" to track)
$
```

```
~/tmp/Alice/test
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    toto.c

nothing added to commit but untracked files present (use "git add" to track)
$
```

```
~/tmp/Alice/test
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    toto.c

nothing added to commit but untracked files present (use "git add" to track)
$
```

```
~/tmp/Alice/test
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    toto.c

nothing added to commit but untracked files present (use "git add" to track)
$
```

Gestion des branches

La principale difficulté de [undefined](#) 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
  master
```

- 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`

Les différents merge

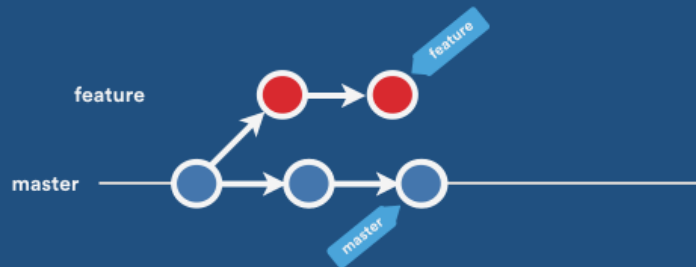
Les exemples suivants sont tirés de :

 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

Explicit **merge**

What is a merge?

A process that unifies the work done in two branches

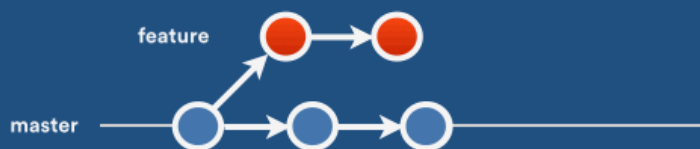


https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/?utm_source=twitter&utm_medium=social&utm_campaign=atlassian_pull-request-merge-strategies-the-great-debate

Implicit merge Via **rebase** Or **fast-forward**

What is a rebase?

It's a way to replay commits, one by one, on top of a branch

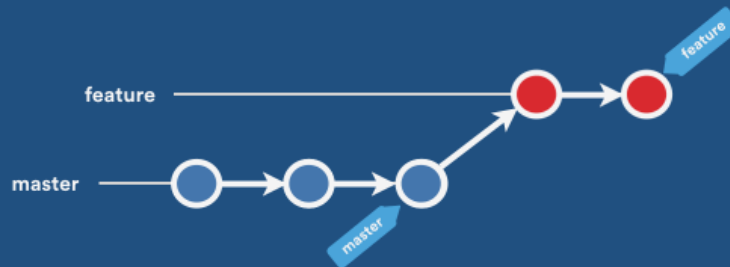


https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/?utm_source=twitter&utm_medium=social&utm_campaign=atlassian_pull-request-merge-strategies-the-great-debate

Implicit merge Via **fast-forward**

What is a fast-forward merge?

It will just shift the
master HEAD

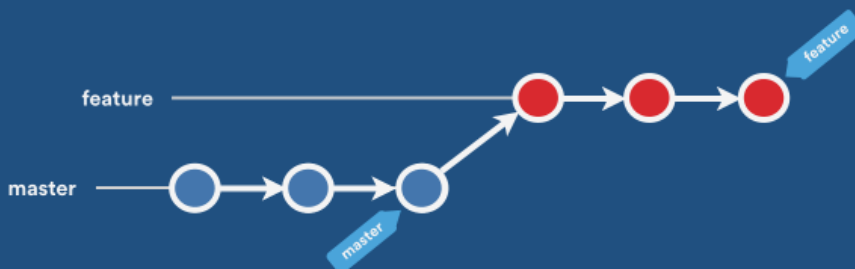


https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/?utm_source=twitter&utm_medium=social&utm_campaign=atlassian_pull-request-merge-strategies-the-great-debate

Squash on merge

What is squash on merge?

It will compact feature commits
into one before merging

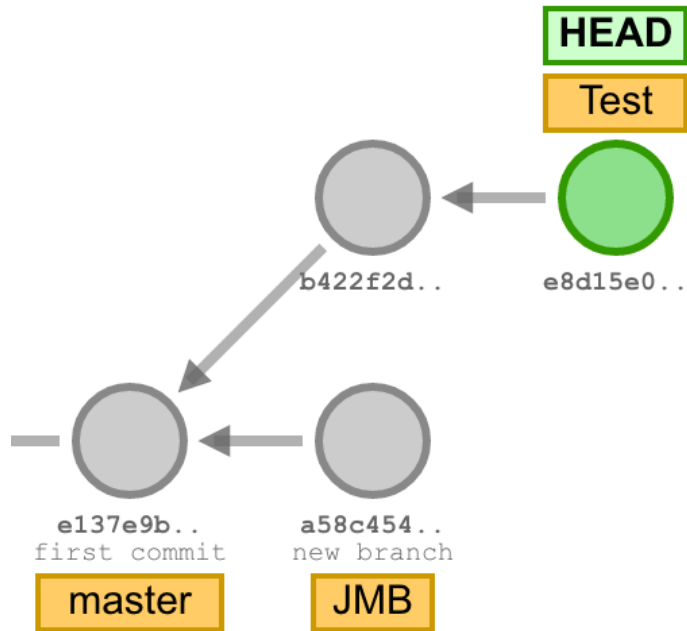


https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/?utm_source=twitter&utm_medium=social&utm_campaign=atlassian_pull-request-merge-strategies-the-great-debate

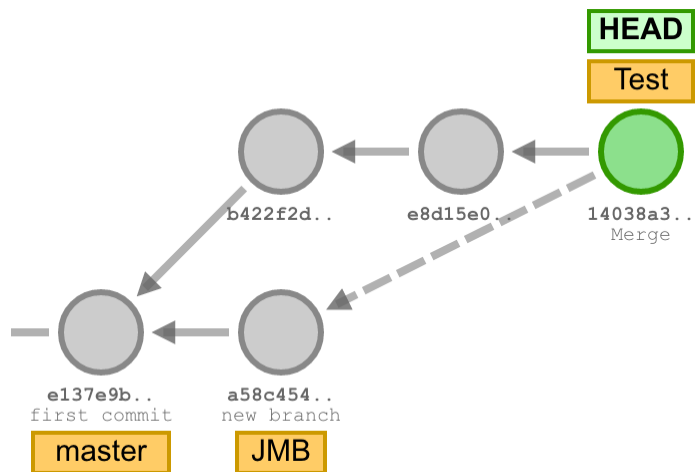
merge VS. rebase

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

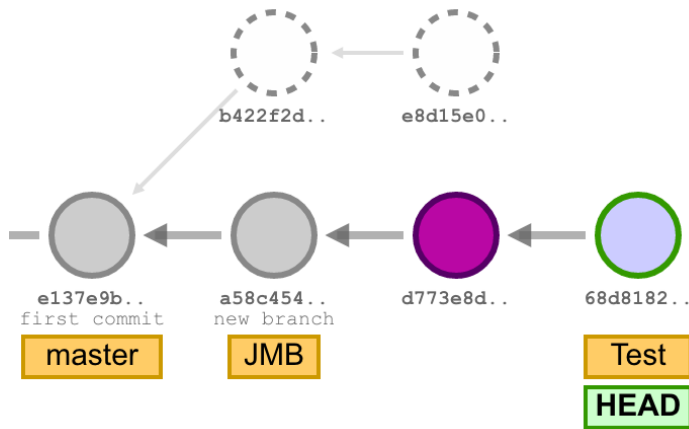
Initial situation:



`git merge JMB:`



`git rebase JMB:`



Gestion des conflits

La principale activité du programmeur qui utilise [undefined](#) en équipe vient de la gestion des **conflits**.

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
Salut monde
=====
hello world!

>>>>>>> other_branch
$ vi toto.txt
$ git commit
```

- 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
- i** Il est déconseillé d'en profiter pour faire une nouvelle modif dans le fichier...

Avec un peu d'aide

- `git diff`
- `git difftool`
 - [DiffMerge](#)

◦ ...

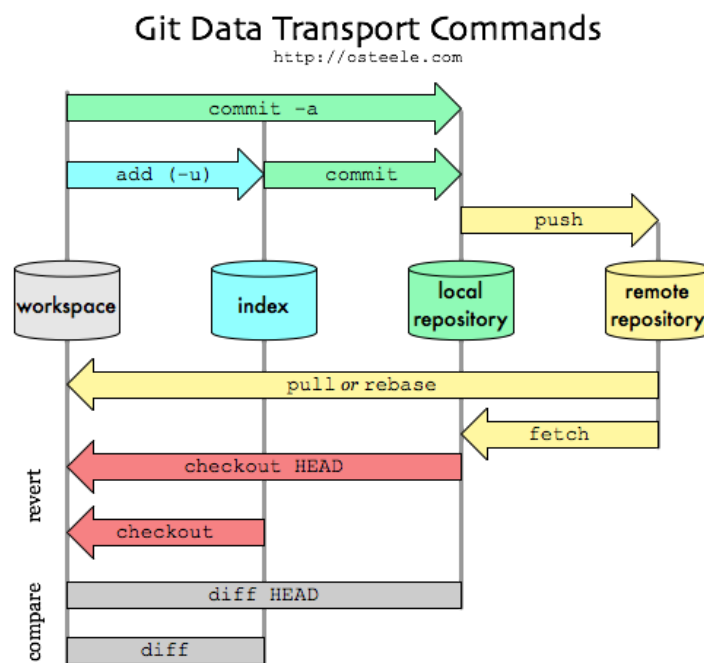
Git avancé : Git-Flow

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

Wrap-up

Résumé des commandes

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



The End