Pour arrêter de galérer avec Git

J’adore Git ! Depuis 5 ans que je l’utilise quotidiennement (http://www.miximum.fr/tag/git), je ne me lasse pas d’admirer la puissance sublime de cet outil, et je ne compte plus les fois ou ma vie fut sauvée par l’une ou l’autre de ces obscures mais miraculeuses commandes. D’ailleurs, n’est-ce pas Aristote qui as dit « Donnez-moi vim et git, et je soulèverai le monde » ? Ce n’est pas un hasard si en Swahili, « Git » signifie « divinité toute puissante à la sagacité du renard, la volupté de l’hippopotame et la virilité du bonobo ».

Je dois pourtant reconnaître que Git n’est pas forcément l’outil le plus abordable qui soit. Toutes ces commandes bizarres ! Toutes ces options apparemment redondantes ! Cette documentation cryptique ! Et ce worflow de travail, qui nécessite 18 étapes pour pousser un patch sur le serveur. Tel un fier et farouche étalon des steppes sauvages, Git ne se laissera approcher qu’avec cisconspection, et demandera beaucoup de patience avant de s’avouer dompté.

C’est surtout après avoir eu l’occasion de donner une formation de 2 jours sur Git récemment que j’ai pu vraiment approfondir certains concepts, résoudre un certain nombre de « WTF ?! » qui restaient en suspens, et réaliser à quel point l’outil était inabordable pour les débutants et les habitués d’anciens systèmes tels que svn (pardon).

Bienvenue

Je suis Thibault, ingénieur web freelance (/a-propos) :

- ingénieur : je résous vos problèmes ;
- web : j'utilise les technologies du web ;
- freelance : vous pouvez me contacter pour profiter de mes services.

Je facilite la vie de mes clients, à coups de développements spécifiques, consulting et formations. Vous pouvez consulter quelques unes de mes réalisations (/category/realisations).

En savoir plus (/a-propos)

Copinage

- DoobleM (http://www.positon.org/)
- Elkorado (http://www.elkorado.com/)
- Palsambleu

Tenez, prenez l'exemple suivant :

Comment j'annule une modification d'un fichier ?

```
git checkout
```

Ok, comment je change de branche ?

```
git checkout
```

Ok, et comment je créé une nouvelle branche ?

```
git checkout
```

Mmm… Ok, et comment je supprime une branche ?

```
git branch -d ma_branche
```

D'accord, et comment je supprime une branche distante ?

```
git push origin :ma_branche
```

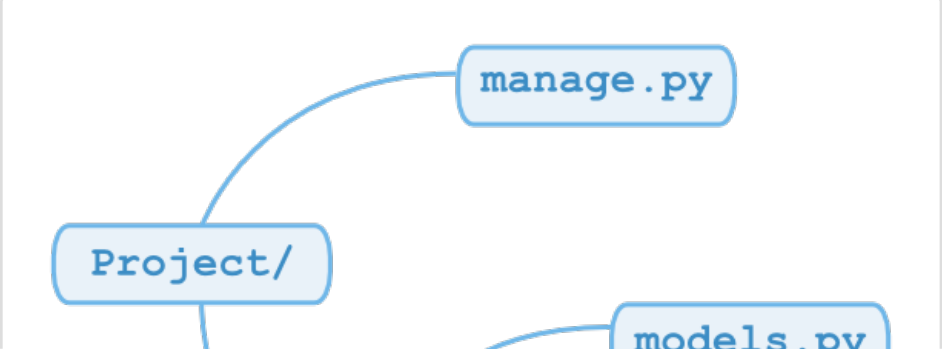
Heu…

À vrai dire, il est trés difficile de percevoir la logique de ce qu'on fait si on n'a pas un minimum de compréhension du fonctionnement interne de Git. De plus, la plupart des commandes sont plutôt bas niveau, ce qui explique leurs effets qui peuvent paraître sans rapports entre eux.

Tâchons de comprendre un peu mieux la bête pour mieux la maîtriser.

Comprendre les zones et le workflow de travail

Imaginons que vous soyez en train de travailler sur un quelconque projet, constitués d'une arborescence de fichiers tout à fait classique.

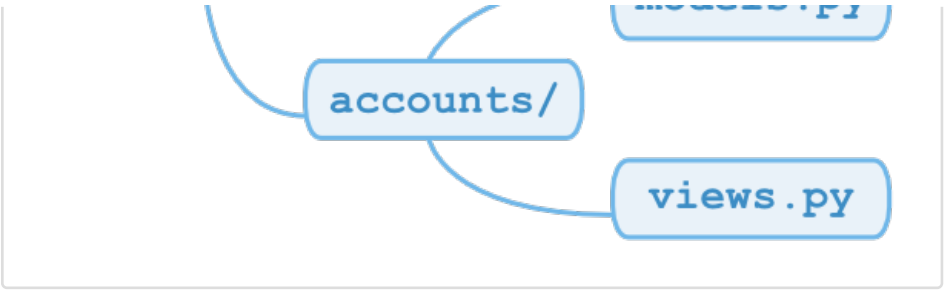


(http://www.palsambleu.fr)

Paris Web (http://www.paris-web.fr/)

Scopyleft (http://scopyleft.fr/)

Sud Web (http://sudweb.fr)

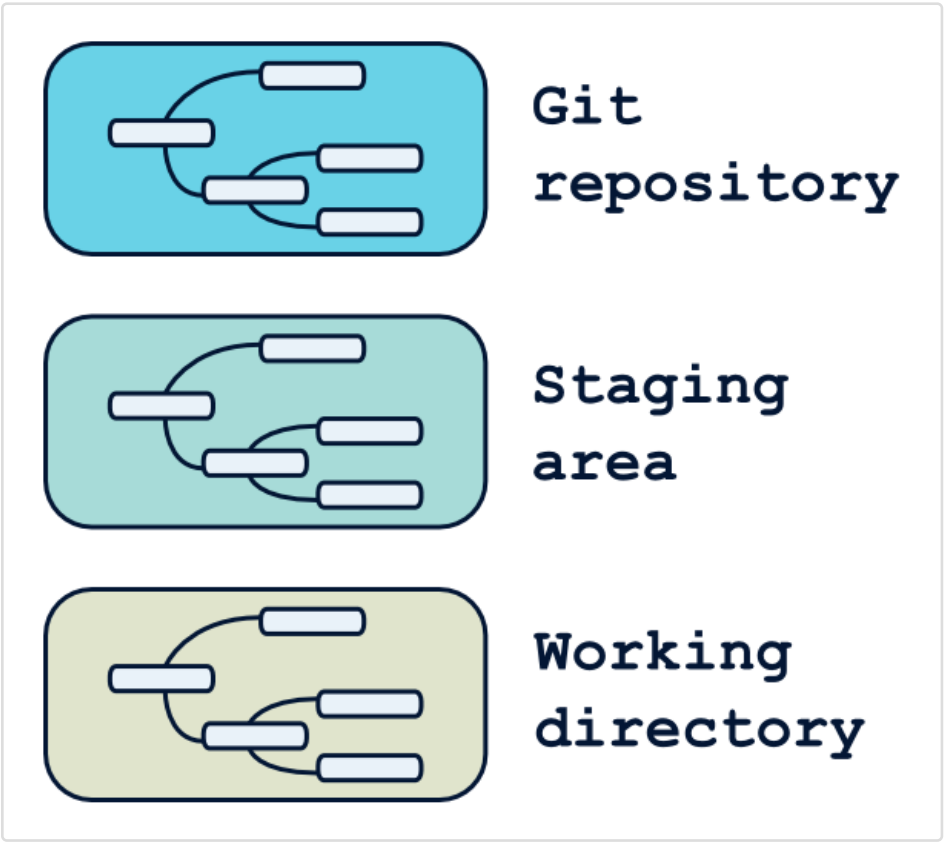


Si votre projet est géré avec Git, on peut grosso-modo considérer que votre arborescence n'est pas stockée une, mais **trois** fois.

D'abord, les fichiers eux-mêmes sur votre disque dur, que vous éditez grâce à votre éditeur préféré (vim, of course (http://vimebook.com/)). C'est votre répertoire de travail, ou *Working Directory* en angliche.

Ensuite, dans une mystérieuse zone spéciale que l'on appelle l'index, ou la zone de staging.

Enfin, dans la base de données de Git est stockée l'arborescence de votre projet telle qu'elle était lors de votre dernier commit.



Pourquoi trois zones, et pas seulement deux ? Quelle est donc cette mystèrieuse « staging area » ?

Qu'est-ce qu'un bon commit ?

Laissez-moi digresser quelque peu pour rappeler qu'un bon commit est un commit *atomique* :

- il ne concerne qu'une chose et une seule ;
- il est le plus petit possible (tout en restant cohérent).

Pourtant, lorsqu'on travaille sur une fonctionnalité, il n'est pas rare d'en profiter pour corriger une petite faute d'orthographe par ci, un petit bug qui trénouillait par là, et on se retrouve avec un répertoire de travail contenant des modifications totalement indépendantes. Ces modifications doivent alors faire l'objet de commits séparés, et c'est à ça que sert la zone de staging : préparer le prochain commit, en y ajoutant ou retirant des fichiers (ou portions de fichiers) sans toucher à votre répertoire de travail.

Certains y verront sans doute un travail superflu bon à satisfaire les instincts pervers des aficionados d'attouchements intimes sur les diptères. Il n'en est rien, et une fois qu'on y a goûté, il est tout simplement impossible de revenir en arrière.

Commandes de base

Le processus de commit avec Git est donc celui-ci :

1. je développe en modifiant / déplaçant / supprimant des fichiers ;
2. quand une série de modification est cohérente et digne d'être commitée, je la place dans la zone de staging ;
3. je vérifie que l'état de ma zone de staging est satisfaisant ;
4. je committe ;
5. et on répète jusqu'à… euh… ben, la fin quoi.

Pour copier un fichier du répertoire de travail vers la zone de staging, on utilise *git add*.

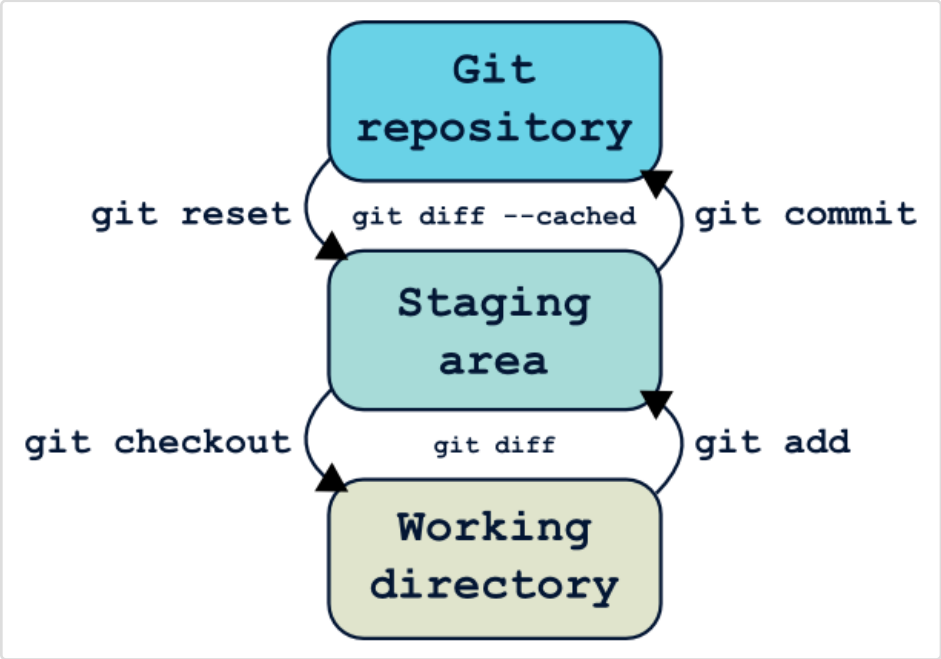
Pour sauvegarder la zone de staging dans le dépôt git et créer un nouveau commit, on utilise *git commit*.

Pour copier un fichier du dépôt Git vers la zone de staging, on utilise *git reset*.

Pour copier un fichier du staging vers le working directory (donc supprimer les modifications en cours), on utilise *git checkout*.

Pour visualiser les modifications entre le répertoire de travail et la zone de staging, on utilise *git diff*.

Pour visualiser les modifications entre la zone de staging et le dernier commit, on utilise *git diff –cached*.



Et comment sait-on quels fichiers sont différents d'une zone à l'autre ? Grâce à *git status* :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   accounts/forms.py
#       modified:   accounts/models.py
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in workin
#
#       modified:   accounts/urls.py
#       modified:   accounts/views.py
#
```

Un arbre de commits

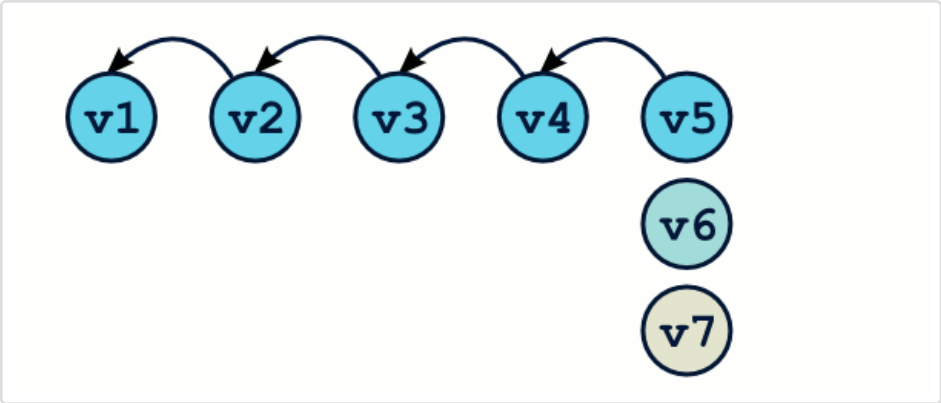
Tout ce pataquès juste pour un commit ? Oui ! Ça peut paraître beaucoup de travail aux habitués de svn (pardon), mais j'affirme que, qu'il travaille seul ou en équipe, un développeur se doit de traiter son historique de projet avec

soin, et de s'assurer de la propreté de chaque commit.

Maintenant, que se passe-t-il lorsque les commits s'enchaînent ? En fait, il suffit de comprendre qu'un commit n'est rien d'autre qu'une structure qui contient :

- des méta-données (auteur, date, etc.) ;
- une référence vers un (ou plusieurs) commit parent ;
- une copie de l'arborescence du projet au moment du commit.

Créer un commit, c'est donc rajouter une entrée dans la base de données Git. Et oui, vous avez bien lu : à chaque commit, Git stocke (de manière compressée) l'intégralité des fichiers qui ont été modifiés depuis le commit précédent. Reconstruire le projet à un instant T à partir de l'historique est donc rapide comme l'éclair.



Avec Git, l'historique du projet n'est ni plus ni moins qu'un graphe, qu'on pourra fouiller grâce à la commande *git log*.

```
$ git log
commit 67d6a5214ad4b259407ec7836b9d729f9f7de731
Author: Thibault Jouannic <thibault@miximum.fr>
Date:   Fri Jul 5 10:45:50 2013 +0200

    create reminders admin module

commit 05ca141b9e982c7d04100c37300da4209305b900
Author: Thibault Jouannic <thibault@miximum.fr>
Date:   Fri Jul 5 10:29:56 2013 +0200

    Create user admin module

commit 0fb74654c708a01bfaec8d552437e9f655bd325d
Author: Thibault Jouannic <thibault@miximum.fr>
Date:   Thu Jul 4 15:46:34 2013 +0200

    upgrade pymill version

…
```

Comprendre les branches

Sous d'autres systèmes comme svn (pardon), la gestion des branches est souvent pénible et laborieuse, ce qui décourage les développeurs qui ne les utilisent que trés occasionnellement (pour ne pas dire jamais). Avec Git, travailler avec des branches est un tel plaisir qu'on aurait tort de ne pas les utiliser. En fait, les branches sont une fonctionnalité basique, pas un truc « avancé » comme je l'entends parfois.

On utilise les branches tout le temps, ou presque. Pour tester une fonctionnalité ; pour isoler un développement un peu long ; pour mettre quelques commits de côté ; pour développer sans péter la branche principale ; pour corriger un bug sans impacter le développement d'une fonctionnalité parallèle. Bref ! il y a pleins de raisons d'utiliser les branches.

Qu'est-ce qu'une branche ? Attention, accrochez-vous, la définition qui va suivre est difficile à comprendre du premier coup.

Une branche n'est qu'une étiquette qui pointe vers un commit.

Quoi ?! C'est tout ? Et oui ! Si vous ne me croyez pas, tapez la commande suivante à la racine d'un dépôt git :

```
$ cat .git/refs/heads/master
67d6a5214ad4b259407ec7836b9d729f9f7de731
```

La branche master (par défaut la branche principale et seule branche d'un dépôt) n'est qu'une étiquette qui pointe vers un commit. C'est un simple fichier qui ne contient rien d'autre que l'identifiant (un hash SHA1) d'un commit. Dans le jargon Git, cette notion de nom référençant un commit s'appelle une « référence ». Un autre exemple de référence nous est donné par les tags.

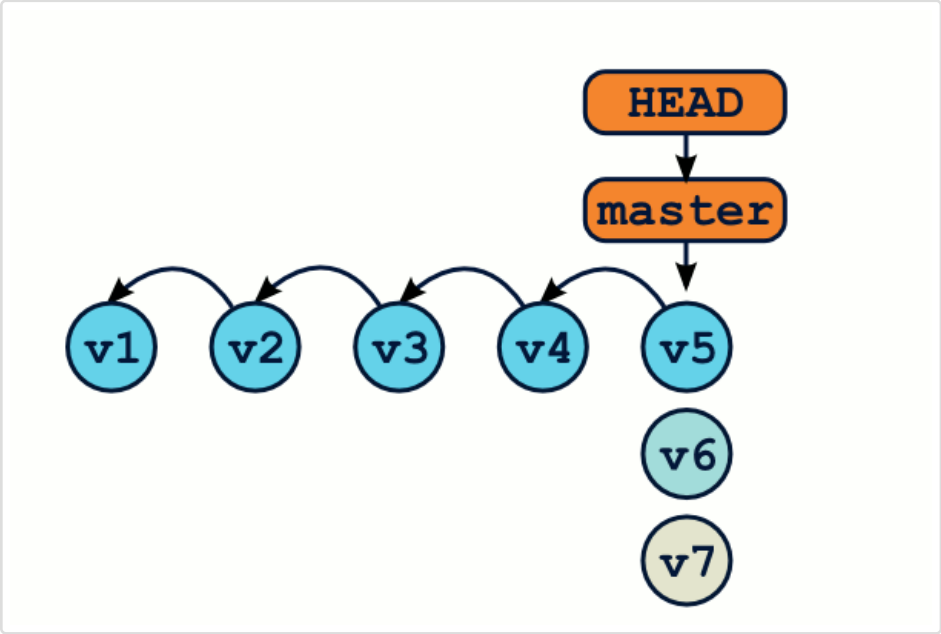
Avec Git, il existe une référence spéciale qui s'appelle *HEAD*. La référence HEAD pointe vers le commit qui sera le parent du prochain commit. C'est clair ? Non ? Vous allez comprendre à la prochaine illustration.

La plupart du temps, HEAD ne pointe pas directement vers un identifiant de commit, mais plutôt vers une branche, e.g *master*.

```
$ cat .git/HEAD
ref: refs/heads/master
```

Quand vous créez un nouveau commit, il se passe ceci :

1. un nouvel objet commit est créé, avec pour parent le commit pointé par HEAD ;
2. la branche pointée par HEAD pointe maintenant vers ce nouveau commit ;
3. et c'est tout.



Manipuler les branches

On va principalement manipuler les branches grâce à deux commandes :

git branch permet de créer, lister et supprimer des branches.

git checkout permet de déplacer la référence HEAD, notamment vers une nouvelle branche.

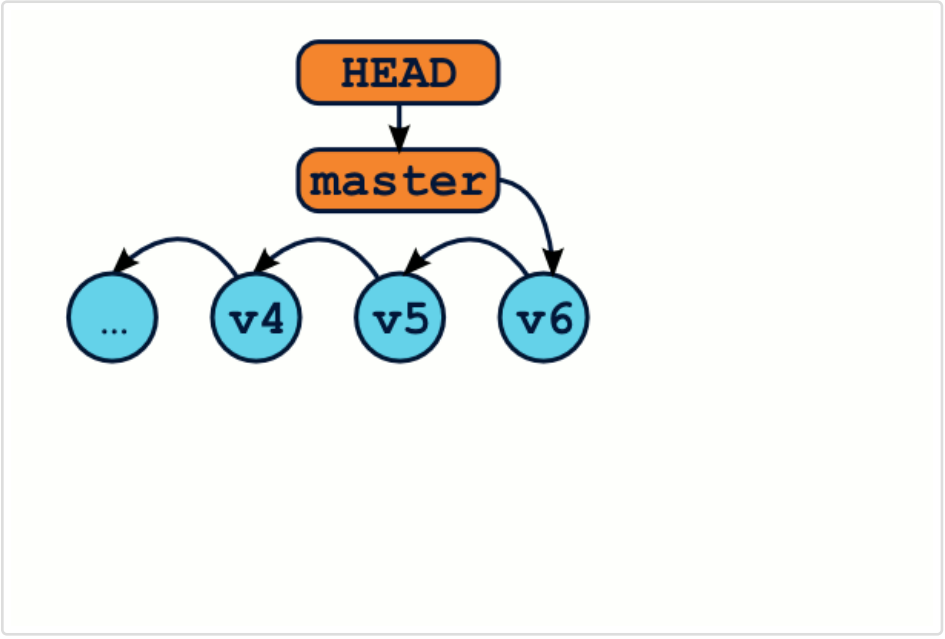
Créer une branche

Voici de manière schématique comment on créé une branche :

```
$ git branch test  # créé la branche "test"
$ git checkout test  # Déplace HEAD sur "test"
```

Notez qu'en général, on utilise le raccourci suivant, qui est trés exactement équivalent aux deux commandes précédentes :

```
$ git checkout -b test
```



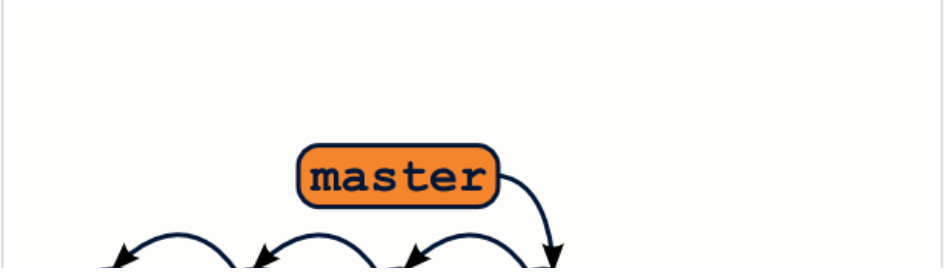
Travailler sur des branches

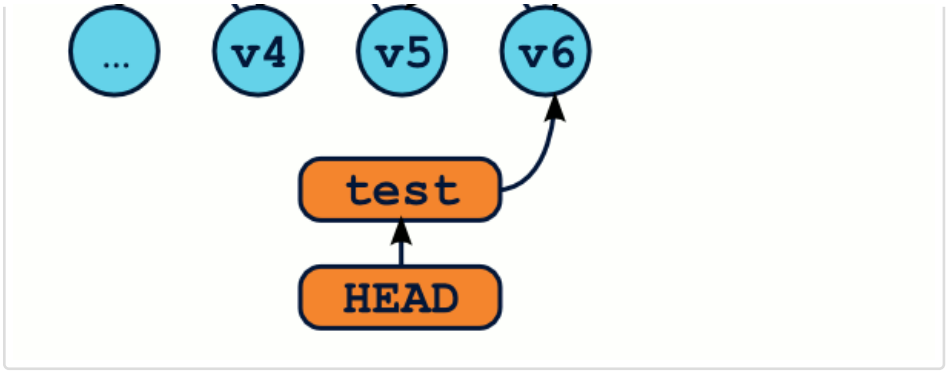
Créer des branches va nous permettre de créer des flots de développement parallèles. Chaque commit sera ajouté à la chaîne des commits de la branche courante. On connaît la branche courante grâce à la commande *git branch*.

```
$ git branch
* master
  test
```

Voici un workflow de travail classique, ainsi que les commandes associées :

```
$ # Je me trouve actuellement sur la branche "test"
$ git commit …  # Je travaille sur ma branche en créant des commi
$ git checkout master  # Retournons sur la branche master
$ git commit …  # je travaille sur ma branche master
```



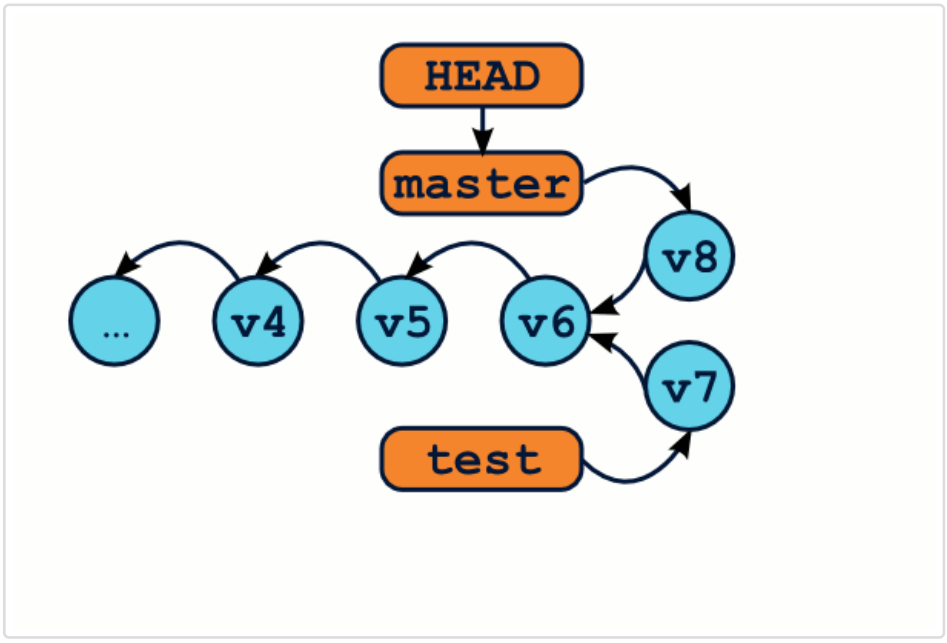


Fusionner des branches

Avoir des branches divergeantes, c'est bien beau, mais il faudra bien fusionner toutes ces modifications dans une seule et même arborescence, n'est-ce pas ? C'est à ça que servent les fusions, ou *merge* dans le jargon.

Lorsqu'on fusionne deux branches, Git va intégrer toutes les modifications contenues sur chaque branche (http://en.wikipedia.org/wiki/Merge_%28revision_control%29#Three-way_merge) dans une seule et même arborescence. Il va alors créer un commit qui aura deux parents. Une délégation de l'UMP aurait réclamé l'obligation pour chaque commit de disposer d'un papa et d'une maman, sans effet jusqu'à présent.

```
$ git checkout master  # On se place sur la branche qui va "recev
$ git merge test  # FuuuuuuuuuuuSion !
$ git branch -d test  # Une fois fusionnée, notre branche ne sert
```



Résoudre les conflits de fusion

Il peut arriver que la fusion soit impossible à réaliser sans intervention manuelle. Ce sera le cas si chaque branche apporte des modifications différentes au même endroit dans le même fichier. On dit qu'il y a conflit.

Dans ce cas là, Git interrompt le merge et insère des marqueurs dans les fichiers conflictuels. Il nous faudra éditer ces fichiers manuellement (ou à l'aide d'une interface spécifique), avant de poursuivre la fusion.

```
Here are lines that are either unchanged fr
om the common
ancestor, or cleanly resolved because only
one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly re
solved or unmodified
```

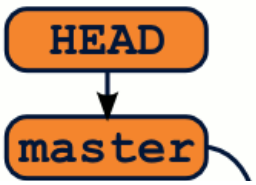
Le protocole précis de résolution est trés bien décrit dans la documentation de la commande *merge*.

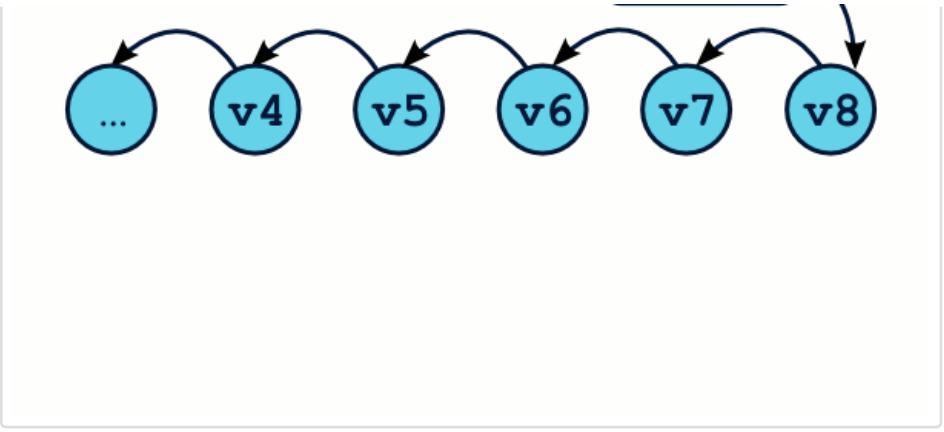
```
$ git help merge
```

J'ai des petits problèmes dans ma plantation

Il est trés utile de comprendre que des branches ne sont rien que des étiquettes qui pointent vers des commits. On s'aperçoit alors qu'il est littéralement possible de créer des branches à n'importe quel moment, même depuis un ancien commit.

```
$ git checkout -b test <old_commit_id>
$ Commit… commit… commit…
```

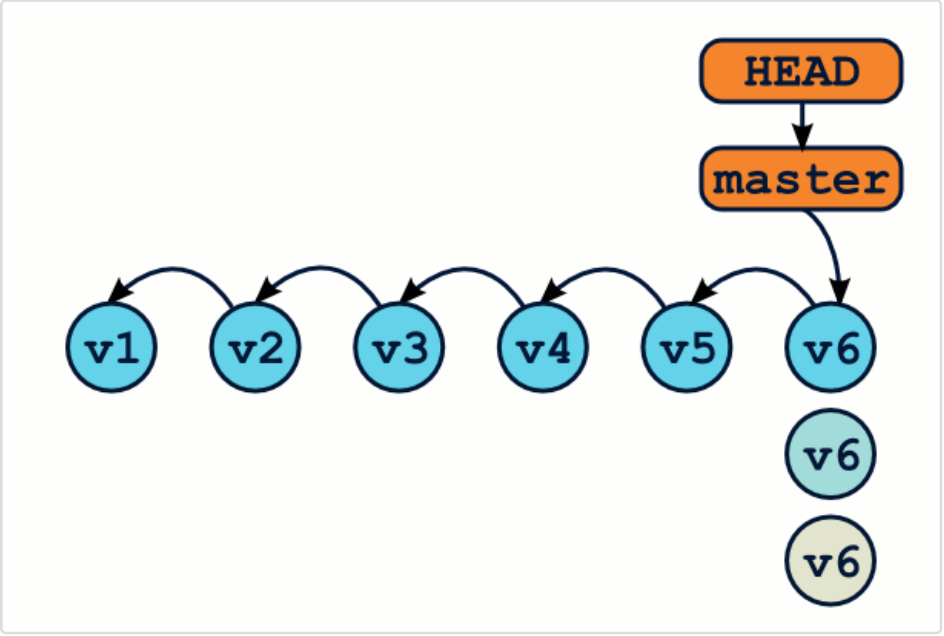




L'état DETACHED HEAD

Jusqu'à maintenant, notre référence HEAD a toujours pointé vers une branche, vous vous rappelez ? Ainsi, la commande *git checkout ma_branche* déplace notre HEAD vers la référence « ma_branche ».

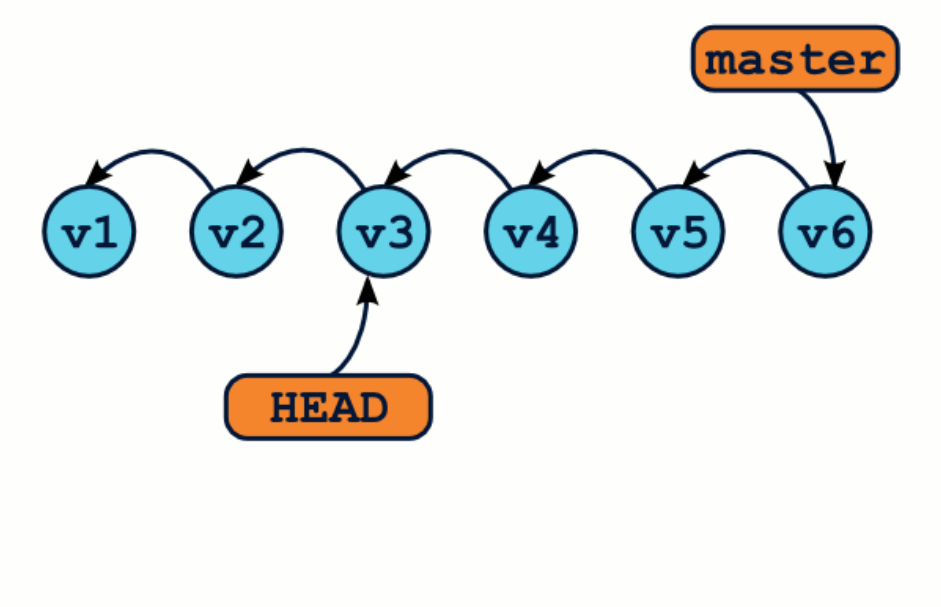
Mais que se passe-t-il si, en lieu et place d'une référence, nous passons directement un identifiant de commit à la commande *git checkout* ?



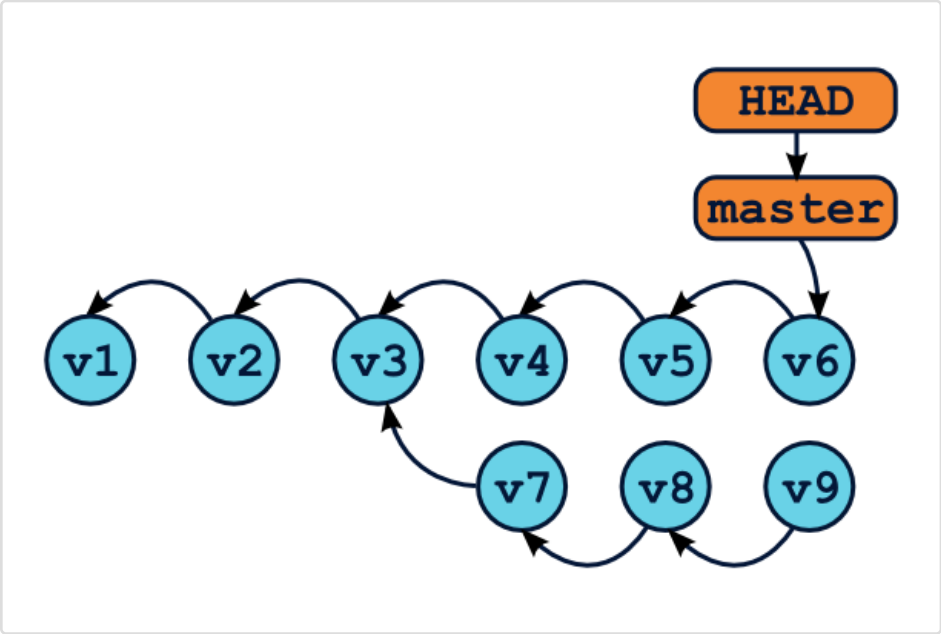
On se retrouve dans ce qu'on appelle l'état *Detached HEAD* : on travaille directement sur un commit, plus sur une branche.

```
$ git branch
* (no branch)
  master
```

L'ajout de nouveaux commits se fera de la manière habituelle.



En revanche, que se passe-t-il si nous utilisons la commande *git checkout* pour retourner sur notre branche *master* ?



OMG ! WTF ! Il semblerait bien que nous ayons perdus des commits ! En effet, la commande *git log* n'affichera que les commits de la branche master (de v1 à v6), mais pas les v7, v8 et v9 ! Et à moins de connaître l'identifiant sha1 exact de ces commits, il semble impossible de les récupérer. Ils ont tout bonnement disparu.

Heureusement, Git dispose d'un outil qui va nous sauver la vie : il s'agit du *reflog*. Git conserve un log de tous les déplacements de la référence HEAD, accessible grâce à la commande *git reflog*.

```
$ git reflog
67d6a52 HEAD@{0}: checkout: moving from 05ca141 to master
05ca141 HEAD@{1}: commit: Create user admin module
0fb7465 HEAD@{2}: commit: upgrade pymill version
8f4c5ba HEAD@{3}: commit: Added log message on new reminder creat
bf15474 HEAD@{4}: checkout: moving from master to bf15474
…
```

Bingo ! La première ligne nous indique la référence du commit juste avant le déplacement de HEAD vers master. Nous pouvons alors créer une branche vers ce commit qui nous permettra de le retrouver plus tard.

```
$ git branch test 05ca141  # Créé une branche, et reste sur maste
```

Si récupérer ces commits ne vous intéresse pas, alors laissez les choses en l'état. Les commits qui ne sont accessibles à travers aucune référence sont régulièrement supprimés par le garbage collector de Git.

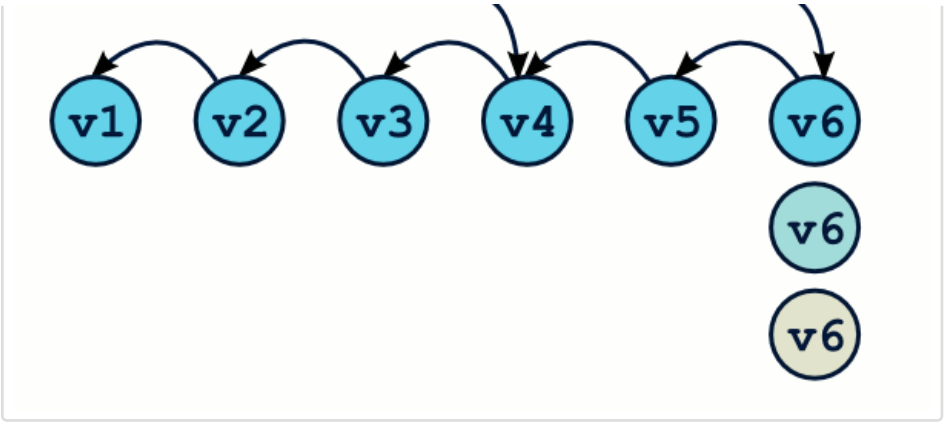
Comprendre cette satanée commande *git checkout*

Si vous avez bien tout suivi, vous avez remarqué que la commande *git checkout* est utilisée à toutes les sauces et semble produire des résultats différents selon les cas.

Il y a en fait deux façons principales d'utiliser la commande *git checkout* :

```
$ git checkout <branch_or_commit>  # 1) Avec un identifiant de co
$ git checkout <branch_or_commit>  <filepath>  # 2) Avec un fichi
$ git checkout <filepath>  # Équivalent à git checkout HEAD <file
$ git checkout  # Équivalent à git checkout HEAD
```

Dans le premier mode d'utilisation (sans spécifier de fichier ou répertoire), Git va simplement déplacer la référence HEAD, et mettre à jour les deux arborescences de la zone de staging et du répertoire de travail. Si vous avez des modifications en cours, elles ne seront pas écrasées, et Git tentera de fusionner la version courante du fichier avec celle correspondant à la nouvelle position de HEAD.



Ainsi, si vous souhaitez récupérer dans votre répertoire de travail votre projet dans l'état ou il était au moment du tag v1.0.1, vous utiliserez la commande *git checkout v1.0.1*, et vous pourrez revenir à l'état précédent grâce à *git checkout master*. On comprend ainsi pourquoi *git checkout* permet de passer d'une branche à l'autre. Par ailleurs, utiliser l'option *-b* permet en plus de créer une nouvelle branche au nouvel emplacement de HEAD.

Le deuxième mode d'utilisation (en spécifiant un fichier ou répertoire) fait grosso-modo la même chose, à quelques exceptions près :

1. la référence HEAD n'est pas déplacée ;
2. la zone de staging n'est pas touchée ;
3. le working directory sera mis à jour à partir de la zone de staging ;
4. l'effet de la commande est limitée au fichier ou répertoire spécifié ;
5. les fichiers en cours de modification seront écrasés purement et simplement.

En gros, la commande *git checkout v1.0.1 accounts* écrase le répertoire *accounts* de votre répertoire de travail avec la version de ce répertoire telle qu'elle était au moment du commit correspondant au tag *v1.0.1*. Clair ?

Voici quelques exemples d'utilisation :

```
$ git checkout .  # Supprime toutes les modifications en cours qu
$ git checkout accounts/  # Supprime toutes les modifications de
$ git checkout v1.0.1 accounts/  # Récupère dans le rép. de trava
```

Un peu tordu ? J'en conviens volontiers. Attendez ! attendez ! Il nous reste à nous attaquer à la commende

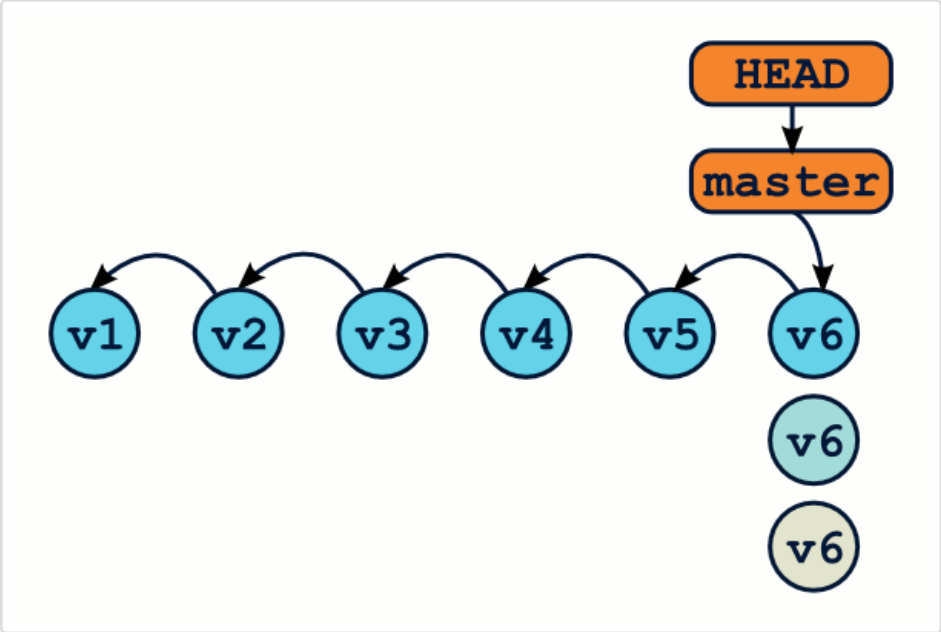
reset.

Comprendre la commande *git reset*

Vous avez aimé *git checkout* ? Vous allez adorer *git reset* ! Le principe de cette commande est grossièrement similaire, sauf qu'en plus de déplacer la référence HEAD, *git reset* déplace également la référence de la branche courante. De même, *git reset* dispose de deux modes d'utilisation : avec ou sans spécifier de chemin de fichier.

Par ailleurs, on peut préciser à la commande quelles zones devront être mises à jour après le déplacement de HEAD.

```
$ git reset v1.0.1 --soft  # Déplace HEAD… et c'est tout !
$ git reset v1.0.1 --mixed  # Déplace HEAD, et met à jour le stag
$ git reset v1.0.1 --hard  # Déplace HEAD, met à jour le staging
```



Attention, contrairement à *git checkout*, *git reset –hard* écrasera votre répertoire de travail même si vous avez des modifications en cours. Il est donc possible de perdre du travail.

Si vous spécifiez un chemin de fichier en plus, alors la commit fonctionnera de manière similaire, à quelques exceptions près :

1. la référence HEAD ne sera pas déplacée ;
2. les modifications seront limitées au chemin de fichier spécifié ;

Quelques exemples d'utilisation :

```
$ git reset  # Équivalent à git reset --mixed HEAD, supprime tout
$ git reset --hard  # Supprime toutes les modifications par rappo
$ git reset --hard v1.0.0  # Supprime tous les commits depuis la
$ git reset HEAD^  # Annule le dernier commit, mais laisse le rép
$ git reset accounts/  # Annule les modifications sous accounts/
$ git reset --hard accounts/  # Cette combinaison d'option est in
```

C'est plus clair ?

Des questions ?

Bon, j'espère que ces quelques éclaircissements vous auront permis d'appréhender Git d'une manière moins empirique. Il parait aussi que je suis capable de donner de chouettes formations sur Git, alors si vous…

- …aimeriez mieux comprendre Git,
- …souhaitez convertir votre entreprise à Git,
- …galérez pendant cette conversion,

n'hésitez pas, contactez-moi. Et si vous avez d'autres questions, n'hésitez pas à les poser, je me ferai (peut-être) un plaisir d'y répondre.

photo by: dollen (http://flickr.com/39804614253@N01/5979760760)

Posté le 09/07/2013 dans Tutos (http://www.miximum.fr/category/tutos) taggé git (http://www.miximum.fr/tag/git).

24 thoughts on "Pour arrêter de galérer avec Git"

> Jade a dit le 09/07/2013 à 17:58 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64800):

Salut,

Super article. Merci pour cette contribution.

PS: Archimède pas Aristote. 😀

> Henri a dit le 09/07/2013 à 21:50 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64802):

Excellent article, qui explique ce qu'il se passe dessous pour comprendre ce qui se passe dessus (ligne de commande).

Super boulot !

> thibault (http://www.miximum.fr) a dit le 10/07/2013 à 09:21 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64804):

@Jade C'est vrai. Et même, je ne suis pas certain qu'Archimède ait parlé de git.

> Sylvain Papet (http://www.com-ocean-web.com) a dit le 10/07/2013 à 11:07 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64805):

Très bon article qui a du te prendre un temps fou, cette vulgarisation à dû bien te casser le cerveau !

Super de faire la promo de VIM en passant, c'est vraiment que niveau environnement de dev il y a pas mieux. Enfin perso je préfère VI qui gère le clavier a plus bas niveau (genre on utilise des lettres pour se déplacer dans le texte, beaucoup mieux que les flèches trop éloignées des touches les plus utilisées), en plus il consomme un peu moins de RAM.

> thibault (http://www.miximum.fr) a dit le 10/07/2013 à 11:11 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64806):

@Sylvain Merci. En fait, vim permet également de

se déplacer en utilisant les lettres. Sinon, ça perdrait pas mal de son intérêt 😀

> Stéphane a dit le 10/07/2013 à 14:43 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64807):

En gros, la commande git **commit** v1.0.1 accounts écrase le répertoire accounts de votre répertoire de travail avec la version de ce répertoire telle qu'elle était au moment du commit correspondant au tag v1.0.1. Clair ?

tu voulais dire git **checkout** ? ou alors j'ai rien compris

> gstr a dit le 10/07/2013 à 15:36 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64808):

Typo dans le paragraphe sur checkout :
« En gros, la commande git commit v1.0.1 accounts écrase le répertoire accounts de votre répertoire de travail avec la version de ce répertoire telle qu'elle était au moment du commit correspondant au tag v1.0.1. Clair ? »
devrait être : « En gros, la commande git **checkout** … » non ?

> thibault (http://www.miximum.fr) a dit le 10/07/2013 à 17:35 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64809):

@Stéphane @gstr Oups ! La typo est corrigée, merci.

> iGor (http://id-libre.org/blogigor) a dit le 11/07/2013 à 06:46 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64810):

En parlant de typo, tout au début (parce que je n'ai pas encore pris le temps de lire l'article que je laisse sous le coude…), il y a un joli « cisconspection ». Ou alors c'était de la publicité déguisée ? 😃

> Michael P. a dit le 11/07/2013 à 16:28 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64811):

Bravo et Merci.

Reste à aborder les checkout sur des dépôts distants, les pull, les push et les fetch.

Une deuxième tournée?

> Marcio a dit le 11/07/2013 à 17:20 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64812):

Super article 😃

Par contre je trouve dommage de ne pas aborder le rebase (http://git-scm.com/book/en/Git-Branching-Rebasing) qui est quand même super cool 😃

> Cyrion a dit le 11/07/2013 à 20:03 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64813):

Excellent article qui serait parfaitement parfait avec des numéros (ou n'importe quoi qui montre la progression) intégrés à chaque frame des gifs animés car comme ils bouclent on ne sait pas où ça commence et ni ou ça termine ! Cordialement 😃

> gui a dit le 11/07/2013 à 21:21 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64814):

Enfin un bon tuto sur Git en français bravo

> Raoul a dit le 14/07/2013 à 14:49 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64819):

Très instructif.
Par contre, si j'ai un projet pré-existant de plusieurs centaines de fichiers que je veux migrer sous GIT, comment faire ?

> thibault (http://www.miximum.fr) a dit le 14/07/2013 à 21:36 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64821):

@Raoul Ça dépend ? Est-ce qu'il faut importer l'historique depuis un autre SGV, par exemple SVN ? Si oui -> http://stackoverflow.com/questions/79165/how-to-migrate-svn-with-history-to-a-new-git-repository (http://stackoverflow.com/questions/79165/how-to-migrate-svn-with-history-to-a-new-git-repository)

Si non :
$ git init
$ git add .
$ git commit

> Bobinours a dit le 17/07/2013 à 08:19 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64829):

Merci beaucoup pour cette vulgarisation qui tombe à point nommé. Il me reste encore pas mal de zones d'ombre, mais c'est déjà une bonne base pour comprendre la suite.

> thibault (http://www.miximum.fr) a dit le 17/07/2013 à 08:27 (http://www.miximum.fr/tutos/1546-enfin-

comprendre-git#comment-64830):

@Bobinours Des zones d'ombres ? Lesquelles ? Ça m'intéresse, ça fera peut-être l'objet d'un nouveau tutoriel.

> Bobinours a dit le 18/07/2013 à 02:34 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64832):

@thibault : comme dit plus haut par Michael P, tout ce qui attrait aux dépôts distants et au travail collaboratif. Ça ajoute un 4ème espace de stockage pour le projet et je crois que ça participe beaucoup à l'incompréhension des débutants.

Le git stash aussi…

Et des exemples pratiques pour faciliter certaines tâches seraient chouettes (Trucs et astuces : lorsque vous voulez faire [...] exécutez la commande « git commande –options » ça vous permettra de [...]).

> Philippe a dit le 19/07/2013 à 20:47 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64838):

Un des meilleurs tutos sur git que j'ai jamais vu, que ce soit en français ou en anglais. Et de loin le plus marrant. Bravo !

Juste un truc : les animations seraient plus claires si la première frame était dupliquée (pour durer plus longtemps).

En tout cas, j'attends la suite avec impatience. 😃

> pointroot (http://www.pointroot.org) a dit le 24/07/2013 à 19:01 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64849):

Top ! Merci

> Devart (http://blog.dev-art.fr) a dit le 25/07/2013 à 17:41 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64850):

Bravo pour cet article !!

> travail à domicile (http://www.lacvtek.com/) a dit le 29/07/2013 à 18:22 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64852):

Merci pour ce partage. Les outils comme GIT ne seraient surement pas faciles d'utilisation pour les freelancers, les chefs d'entreprise en herbe, les travailleurs indépendants sans ce genre de mode d'emploi détaillé. L'esprit entrepreneurial ne peut qu'être boosté.

> Eric Bouchut (http://Ericbouchut.com) a dit le 31/07/2013 à 13:17 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64857):

Merci Thibault pour cet article didactique intéressant.

Il y a me semble-t-il, une typo dans le paragraphe « Comprendre cette satanée commande git checkout »:

> Le deuxième mode d'utilisation (en spécifiant un fichier ou répertoire) fait grosso-modo la même chose, à quelques exceptions près :
> la référence HEAD n'est pas déplacée ;
> la zone de staging n'est pas touchée ;
> le working directory sera mis à jour à partir ~~de la zone de staging~~ du dépot git;
> [...]

> cyp (http://www.rouquin.me) a dit le 02/08/2013 à 16:28 (http://www.miximum.fr/tutos/1546-enfin-comprendre-git#comment-64860):

chouette article merci !

Les commentaires sont fermés.

Article suivant → (http://www.miximum.fr/humour/1631-la-police-de-la-civilite)
← Article précédent (http://www.miximum.fr/tutos/1516-integrer-des-formulaires-de-paiement-sur-son-site-django-avec-paymill)

À propos

Thibault Jouannic
Ingénieur web freelance

Adresse

3 Avenue de Palavas
34070 Montpellier

Me contacter

Twitter (http://twitter.com/thibaultj)

EI Profession Libérale
Siret 51469122900017

thibault@jouannic.fr
(mailto:thibault@jouannic.fr)
06 30 47 30 50
(tel:0630473050)

Github (https://github.com
/thibault)
Hacker News
(http://news.ycombinator.com
/user?id=thibaultj)
Stackoverflow
(http://stackoverflow.com
/users/665797/thibault-j)