**Git en 2-2**

Aucun développeur ne peut ignorer l’existence de Git, le célèbre outil de versionning de Linus Torvalds. Que l’on code seul ou en équipe, dès que l’on commence à travailler sur des projets d’une certaine taille, on ne peut se passer d’un bon logiciel de gestion de version. Passons donc rapidement – en 2-2 – en revue les commandes les plus utiles de Git.

Cet article ne vise pas à enseigner le fonctionnement de Git mais permet d’avoir un rappel des commandes répondant aux besoins les plus courants. Si vous souhaitez apprendre à vous servir de Git, je ne peux que vous conseiller [l’excellent livre Git Pro](http://git-scm.com/book/fr/v2), intégralement disponible en ligne sous licence Creative Commons.

**Plan de l’article**

* [Utilisation courante](https://buzut.fr/git-en-2-2/#workflow)
* [Modifications, changements et annulation](https://buzut.fr/git-en-2-2/#revert)
* [Le git add interactif](https://buzut.fr/git-en-2-2/#interactive)
* [Voyage dans le temps](https://buzut.fr/git-en-2-2/#timetravel)
* [Utilisation des branches](https://buzut.fr/git-en-2-2/#branching)
* [Les modules (submodules)](https://buzut.fr/git-en-2-2/#modules)

**Flux de travail normal**

# Cloner un projet

git clone [url]

# Initialiser un dépôt

git init

# Suivre des fichiers ou les ajouter à la staging area

git add fichier1 fichier2

# retrancher un fichier du staging (défaire le git add)

git reset HEAD fichier

# Voir le statut

git status

# Visualiser ce qui a été modifié mais pas encore indexé

git diff

# visualiser les différences entre un commit et l'état actuel ou un autre commit

git diff commithash [commithash2]

# idem que ci-dessus mais seulement pour les fichiers précisés

git diff commithash [commithash2] fichier1 [fichier2 …]

# Demander à git d'afficher les diff des fichiers ajouté à la staging (avec git add)

git diff --staged

# Visualiser les modifications indexées qui feront partie de la prochaine validation

git diff --staged

# Enregistrer "commiter" les modifs

git commit

# Commiter sans passer par l'éditeur de texte

git commit -m "message de commit"

# Tout comiter sans passer par add

git commit -a

# Placer un tag (lightweight)

git tag 1.2.0

# Placer un tag sur un commit antérieur

git tag 1.1.4 ffaae76

# push n'envoie pas les tags par défaut

# pour pusher les tags

git push --tags

# Voir l'historique des commits (l'option --all permet de voir l'ensemble des branches)

# il est possible d'appeler l'option --graph pour rendre ça plus visuel

# on peut préciser un fichier pour n'obtenir que les commits relatifs à ce dernier

git log [--all]

# effectuer une recherche dans les commits (-g pour toutes les branches)

git log [-g] --grep=STRING

# rechercher dans le code en incluant l'historique

# (pratique pour retrouver quelque chose qui n'existe plus dans le HEAD)

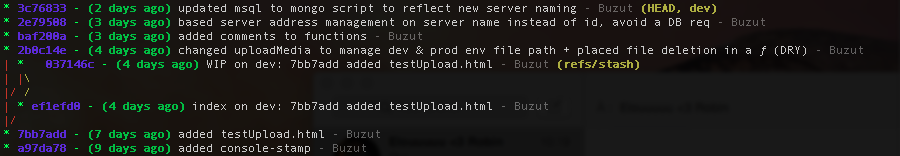
git grep 'votre recherche' $(git rev-list --all)

Étant donné que la commande log s’avère vite lourde dès que l’on souhaite afficher pas mal d’infos de manière propre, voici un petit alias que vous pouvez ajouter dans votre .gitcongif, lequel se trouve en général dans ~/.gitconfig et qui vous permet d’avoir une vision sympa de votre historique :

[alias]

logall = log --graph --abbrev-commit --decorate --date=relative --format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold yellow)%d%C(reset)' --all

Voici ce que ça donne lorsqu’on invoque la commande git logall :



Tout dépend de la manière dont vous commentez vos commit, mais j’aime bien connaître également les fichiers concernés par le commit, dans ce cas, vous pouvez rajoutez à la commande ci-dessus, juste après --graph, --name-only.

**Le staging interactif**

Vous le savez, les [commits doivent être atomiques [en]](https://www.freshconsulting.com/atomic-commits/) autant que faire se peut. Dans un monde idéal, on travaillerait donc sur un point précis, on le commit puis on passe à la fonctionnalité ou au fix suivant. Il est inutile de préciser que les commits du genre fixed login error, added email validation, changed background color and better i18n support ne devrait pas exister !

Bien entendu, ce monde idéal n’existe pas car nos cerveaux, et par voie de conséquence, notre workflow, ne sont pas dans une logique linéraire. Ainsi, on travaille sur un point particulier, puis, dans le même fichier, on remarque qu’un autre point pourrait être amélioré, et hop, deux modifs **distinctes** dans le même fichier.

Je confesse qu’il m’est arrivé de faire tout un tas de bidouille avec des copier-coller dans des fichiers temporaires pour valider mes modifications en plusieurs commits distincts. Mais ça, c’était avant… que je ne découvre la commande magique git add -i.

Elle permet, parmi d’autres choses, de *stager* (c’est à dire git add fichier) seulement une partie d’un fichier. De la sorte, vous faites un premier commit sur vos modifications conernant un point A, puis dans un second temps le point B. RÉ-VO-LU-TION-NAIRE !

En plus, lorsque vous avez beaucoup de fichiers à commiter et qu’un git commit -am n’est pas envisageable, la commande vous offre une alternative agréable et efficace à git add fichier1 fichier2 fichierN.

Je ne vais pas tout vous expliquer ici car ce serait un peu long, mais dans mon infinie bonté, je vous livre directement l’[url du chapitre Git Book](https://git-scm.com/book/en/v2/Git-Tools-Interactive-Staging) qui va bien. (PS : si je viens de vous changer la vie, vous pouvez m’envoyer une boite de chocolat).

**Modifications, changements et effacements de fichiers**

# Effacer un fichier (arrête le suivi et efface la fichier du répertoire de travail)

git rm monfichier

# Arrêter de suivre un fichier mais ne pas l'effacer

git rm --cached monfichier

# Déplacer des fichiers

git mv fichierSrc fichierDest

**Annuler et modifier des commits**

# Modifier le dernier commit

git commit --amend

# Réinitialiser un fichier modifié (avant commit)

git checkout -- monfichier

Comme souvent avec git, --amend possède plusieurs comportement selon le contexte. Il est donc de bon ton de préciser tout cela. Si vous n’avez aucun fichier indexé, dans la *staging area* (ie. vous n’avez pas fait de git add depuis le dernier commit) ; la commande vous permet simplement d’éditer le dernier message de commit.

Cependant, si vous avez ajouté des fichiers à l’index, ces modifications seront directement ajouté au dernier commit (en plus de pouvoir modifier le message). Bien pratique lorsqu’on a oublié de commiter un détail.

**Rebase, l’effet papillon !**

La commande rebase permet de « réécrire l’histoire », comme le définit la documentation. Grâce à cette commande, il est possible de modifier d’anciens messages de commit, de supprimer des commits, de merger plusieurs commits en un seul etc. Cette commande est ultra puissante. Elle possède un mode interactif qui permet d’effectuer les modifications aisément puisque tout est expliqué. Pour lancer la commande, il vous suffit de préciser jusqu’à quel commit (ou combien de commits) vous voulez remonter et hop.

Pour info, puisqu’en général on a besoin de modifier un commit en particulier, il faut remonter au commit précédent celui-ci pour pourvoir le modifier !

# lancer rebase en mode interactif et remonter x commits dans le temps

git rebase -i HEAD~5

# ou

# précisez toujours le commit précédant de celui à modifier

git rebase -i e581386

# ci-dessous, un exemple des commandes proposées

pick 9d42307 deleteUser calls userMediaModel instead of calling mediactrl

pick da199cc moved part of deleteMedia logic to model

pick 2968bd0 use BadRequestError intead of custom mongoose err in models

# Rebase 5607db4..2968bd0 onto 5607db4 (3 command(s))

#

# Commands:

# p, pick = use commit

# r, reword = use commit, but edit the commit message

# e, edit = use commit, but stop for amending

# s, squash = use commit, but meld into previous commit

# f, fixup = like "squash", but discard this commit's log message

# x, exec = run command (the rest of the line) using shell

# d, drop = remove commit

Comme vous le constatez, par défaut, tous les commits sont pick, c’est à dire qu’ils sont laissés tels quels. La plupart du temps, vous voudrez remplacer pick par r, e ou d afin de respectivement éditer le message, modifier les fichiers du commit ou complètement le supprimer.

Le cas que je trouve le plus intéressant (du moins que j’utilise le plus souvent et qui est d’une extrême puissance) est le edit. Dès lors que vous sauvegardez et que vous quittez votre éditeur de texte, git vous ramène à l’état du commit que vous voulez éditer. Là, vous effectuez vos modifications, un petit git add pour la forme, puis, lorsque vous avez terminé vos changement git rebase --continue.

Si vous n’aviez pas marqué d’autres commits pour édition, vous voilà revenu au dernier état, avec les éditions du rebase !

Pour en savoir plus sur les possibilités ultra complexes : [RTFM !](http://www.git-scm.com/book/en/v2/Git-Tools-Rewriting-History)

**À la bourrin…**

Je dis bourrin parce qu’il arrive de devoir employer les grands moyens. Cela dit, si les commandes existent, elles sont faites pour être utilisées…

# Annuler dernier commit

git reset HEAD~

# Tout ramener (répertoire de travail inclu) à l'état du commit X

git reset --hard X

Allez à [reset démystifié](http://git-scm.com/book/fr/v2/Utilitaires-Git-Reset-d%C3%A9mystifi%C3%A9) pour comprendre les mécanismes de reset.

Une autre situation se présente parfois : vous vous rendez compte au bout d’un certains nombre de commits que vous avez inclu un fichier sensible (par ex. une clef ssh ou un fichier de conf avec le mot de passe de la bdd…). C’est moche ! Première chose, on va ajouter ce fichier dans le .gitignore pour ne pas que l’erreur se reproduise. Ensuite on va, un peu comme avec rebase, réécrire l’histoire en enlevant des objets. Nous allons pour cela utiliser filter-branch :

git filter-branch --index-filter \

'git rm --cached --ignore-unmatch config/prodConfig.js' \

--tag-name-filter cat -- --all

Au cas où vous n’exécutez pas la commande pour la première fois et que git vous sort une erreur du style Cannot create new backup. A previous backup already exists in refs/original/, il faudra forcer la commande avec l’option -f : git filter-branch -f […].

**Voyager dans le temps**

Nous avons vu, notamment avec reset et rebase que nous pouvions modifier l’histoire. Bien souvent, cela peut s’avérer dangereux et inutile – vous connaissez l’effet papillon ! Pourtant, nous voudrions parfois pouvoir restaurer notre projet, ou un fichier, à un état antérieur pour pouvoir y faire des tests. C’est tout à fait possible.

On va pour cela encore faire appel à checkout. La particularité de cette manipulation est qu’on détache HEAD de la branche sur laquelle on se trouve pour la faire pointer vers un commit en particulier. En d’autres termes plus familiers, on modifie notre « pointeur », non plus pour qu’il reflète le temps présent, mais un *snapshot* qu’on lui désigne.

On peut alors se promener dans notre projet et y faire des modifications sans risque, même en y commitant des changements ! Afin de conserver ces modifications, on pourrait créer une nouvelle branche à partir de l’état « détaché ». Enfin, lorsque l’on veut « raccrocher » HEAD, il faudra de nouveau faire un checkout. Démonstration :

# on visionne le projet à son état lors du commit f654646

git checkout f654646

# pour revenir dans le présent

git checkout uneBranche

On peut également ramener un fichier à l’état d’un commit antérieur. Seulement, dans ce cas, HEAD n’est pas détaché, et dans l’éventualité ou vous faites des modifications au fichier et/ou que vous le commitez, cela est pris en compte. Néanmoins, vous pouvez décidez, après tests, que l’ancienne version et/ou vos modifications ne font pas l’affaire. Dans ce cas, vous utilisez checkout comme vous en avez l’habitude pour rejeter les modifications d’un fichier. Démonstration :

# je ramène server.js à un état antérieur

git checkout f654646 server.js

# après quelques tests, je ne suis pas satisfait

# j'efface mes changements et retrouve ma version d'avant le checkout

git checkout server.js

Il est parfois un peu déroutant d’avoir tant d’usages à la commande checkout. Cependant, elle est vraiment très puissante et sert dans de nombreux cas.

**Travailler avec les branches**

Les branches ont une utilité bien précise. Par défaut, vous êtes sur la branche master (vous pouvez le constater en faisant git branch. Admettons que vous travailliez sur une nouvelle fonctionnalité, et tout d’un coup, vous devez corriger un bug qui n’a rien à voir avec ladite fonctionnalité; où vous avez un éclaire de génie et vous commencez à travailler sur une autre fonctionnalité en parallèle; où vous vous rendez compte qu’il y a une fonction que vous devez absolument re-factoriser parce que vous êtes en train de dupliquer du code comme un bourrin… Bref les exemples ne manquent pas.

Dans tous ces cas, il vous faut une branche. Un conseil presque extrême (surtout quand on commence et que l’on a tendance à ne pas trop ce servir des branches) est de ne jamais travailler sur la master. Pour chacun des cas cités en exemples ci-dessus, il vous faudrait une branche différente. Ainsi, vous avez commencé à écrire une fonction, vous vous interrompez pour corriger un bug et vous commitez celui-ci, votre commit ne contiendra pas la fonction inachevé que vous avez abandonné. Bref, voyons les commandes :

# Lister la branche active et toutes les branches

git branch

# Créer une nouvelle branche

git branch newBranch

# Supprimer une branche

git branch -d vieilleBranche

# Se déplacer sur myBranch

git checkout myBranch

# renommer une branche

# l'ancien nom est facultatif si on veut renommer la branche courante

git branch -m [ancienNom] nouveauNom

Il est tout indiqué de commiter le travail en cours avant de passer d’une branche à une autre. Au cas où vous ne vouliez pas commiter un travail non fini simplement pour changer de branche, git a pensé à vous, il est possible de mettre votre travail en cours dans une « file d’attente » !

# C'est le commit du travail non terminé

git stash

# Permet de voir la file d'attente en cours

git stash list

# Permet de rétablir un des états en attente,

# par défaut, git stash apply revient à l'état le plus récent.

# En nommant l'état en question, on peut revenir à un état antérieur.

git stash apply [stash@{n°}]

# il faut parfois nettoyer la liste de stash

# enlever le dernier élément [ou un élément donné]

git stash drop [stash@{n°}]

#effacer la liste

git stash clear

Une fois l’ensemble des modifications terminées, il suffit de « merger », fusionner, la branche annexe avec la brancher master. Pour cela, on retourne sur la branche master (ou sur celle avec laquelle on veut fusionner) avec checkout puis :

# Fusionne la branche "myBranch" avec la branche sur laquelle on se trouve

git merge myBranch

Il est souvent pratique de savoir quelles branches ont été mergées avec celle dans laquelle on se trouve, pour cela il y a l’option –merge :

# Faffiche les branche déjà mergées avec la branche courante

git branch --merge

# et l'inverse, celles non mergées

git branch --no-merge

**Les remotes**

Savoir que son code est pushé sur une machine distante, c’est aussi assez sécurisant. Et que ce soit pour collaborer ou pour simplement ajouter votre pierre à l’édifice de l’open source, Git est fait pour fonctionner avec des remotes !

# ajouter une remote

git remote add origin url-de-la-remote

# il est aussi possible d'en avoir plusieurs

# ex github + votre serveur git perso

# pour en ajouter une autre

git remote add alt url-de-la-remote

**Les git submodules**

Pour comprendre de quoi il s’agit, commençons par expliquer le use case. Vous développez un module réutilisable, que vous voulez intégrer dans divers projets. Par exemple un petit système de notification.

Soit vous copiez le code dans chaque projet et vous le commitez, mais vous dupliquez le code partout, vous avez des versions différentes etc. Pas pratique. Soit vous ajoutez ce module dans le .gitignore et vous le clonez dans le projet. À chaque modif, il faudra bien penser à pull, pour récupérer la dernière version, push si vous effectuez des modifs en local etc. Bref, la galère. Et puis les gens qui cloneront le repo devront se farcir la dépendance manuellement (.gitignore oblige…).

Sinon il existe les git submodules. Vous dites à git de référencer un module depuis un autre repos. Lors d’un clone, le repo tiers sera récupéré avec, si vous effectuez des modifs en local, vous pourrez les pousser depuis le répertoire du submodule et vous pourrez facilement merger depuis la remote. Extra non ? Pour ce faire, rien de bien sorcier :

git submodule add https://mygit.com/username/repo

Lorsque vous clonez un repo qui contient des submodules, par défaut ces derniers ne sont pas automatiquement importés. Les répertoires de ces derniers sont créés mais vides. Deux commandes seront nécessaires pour respectivement initialiser le fichier de config et récupérer lesdits submodules.

git submodule init

git submodule update

# all gooood

Pour creuser ça plus en profondeur, il y a un chapitre dédié dans [Git Pro](https://git-scm.com/book/fr/v2/Utilitaires-Git-Sous-modules).

Bien entendu, cette cheat sheet n’est pas exhaustive, n’hésitez pas à me faire part de commandes qu’il faudrait ajouter !