# Git overview

Petit rapport sur git avec de nombreux exemple de commande, des plus courantes aux plus spécifiques.

Table des matières

[Git overview 1](#_Toc191042)

[Principe de git 2](#_Toc191043)

[Commandes les plus courantes 2](#_Toc191044)

[Config - Initialisation des variables locales 2](#_Toc191045)

[Add et commit 3](#_Toc191046)

[Log – Afficher des commits 4](#_Toc191047)

[Status & diff – Visualiser les modifications 4](#_Toc191048)

[Rm et mv – Supprimer, renommer ou déplacer des fichiers 5](#_Toc191049)

[clone – Récupérer un projet GitHub 5](#_Toc191050)

[Fetch & Merge – Mettre à jour depuis GitHub 5](#_Toc191051)

[Push – Mettre mes modifications sur GitHub 6](#_Toc191052)

[Les branches 6](#_Toc191053)

[Branch 7](#_Toc191054)

[Checkout 7](#_Toc191055)

[Renommer ou supprimer une branche 8](#_Toc191056)

[Merge 8](#_Toc191057)

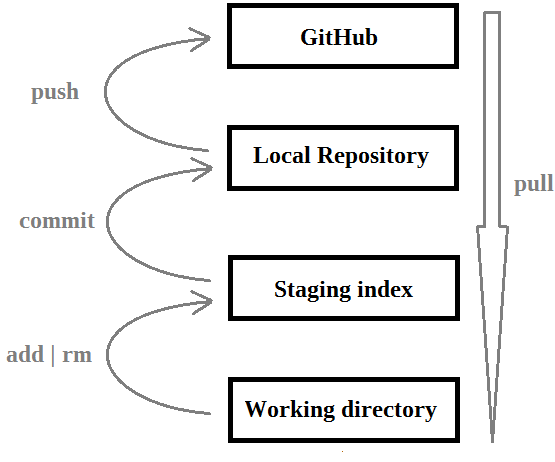
[Stash 10](#_Toc191058)

[Visualisation 10](#_Toc191059)

[Cas d’utilisations 11](#_Toc191060)

[Les commandes stash 12](#_Toc191061)

## Principe de git

Le meilleur moyen selon moi de visualiser git est de la faire au travers de ce schémas, les commandes afficher seront toutes expliquée :

* Le working directory correspond à notre espace physique de travail (le répertoire par la chaise)
* Le staging index est l’espace intermédiaire entre le repository et le working index. On va pouvoir y mettre nos modifications, pour les commit toutes ensembles.
* Le local repository représente l’image de notre travail sauvegardé en une suite de commit
* GitHub est une plateforme d’hébergement de liste de commit.

En effet, sous git, hormis le working directory **tout est commit**, lorsque l’on effectue un push on n’envoie pas le nouveau code mais bien **la liste des modifications** effectuées.

Chaque commit est identifié par un **SHA** qui est une longue suite de caractère générée aléatoirement.

## Commandes les plus courantes

### Config - Initialisation des variables locales

Afficher les config locales :

Git config –list

Les variables locales sont utilisée par git lors de l’exécution de commande, nous utilisons un paramètre :

* Soit : --system qui permet de modifier la viriable pour tous les utilisateurs du PC
* Soit –global qui effectue la modification uniquement sur l’utilisateur courant.

Liste des variables utiles :

* user.name, user.email : sont les deux variables indispensables
* core.editor : permet de sélectionner l’éditeur de texte que git va utiliser
* color.ui true : ajoute des couleur aux affichages.

Exemple :

Git config –global user.name LavergneC

### Add et commit

Il s’agit sans doute des commandes les plus utilisée concernant le dépôt local. L commande Add permet d’ajouter les modifications effectuer sur un fichier à notre ‘staging index’ ou d’y d’ajouter un fichier non tracké. La commande s’écrit sous la forme : git add <fichier>. Le <fichier> peut être remplacé par un point, dans ce cas toutes les modifications sont ajoutées.

Exemple :

Git add Src/main.c

La commande commit va prendre la liste les modifications présentes dans le stagin index et les envoyer dans notre repository. Un commit peut se voir sous forme d’un paquet remplie de modification. Le paramètre -m que l’on mettra **toujours** (sauf pour un merging git s’en charge…) donne le nom à notre paquet.

A utiliser avec parcimonie : La commande git commit -am ʺmessageʺ va effectuer d’un coup un add et un commit. Cette dernière commande ne fonctionne qu’avec les changements de type « modifier ».

Exemple :

Git commit -m ʺAjout de l’init GPIOʺ

Bonnes pratiques :

* Un commit doit porter sur un sujet en particulier
* Le message doit être clair est précis, cela rendra un grand service soit à vous du futur soit à la personne à qui vous demanderait de l’aide (ces deux personnes étant sympathique) lorsque des problèmes surviendront.

### Log – Afficher des commits

Cette commande en elle-même est très simple mais les options associées sont très riche. La commande permet d’afficher des commits. Comme un projet contient souvent beaucoup de commit on utilisera dont l’option -<n> avec n le nombre de commit que l’on veut afficher (en commençant par le dernier).

Quelques options :

* --oneline : Chaque commit est afficher sur une seule ligne
* --since date : afficher des commit depuis une date donnée
* En vrac : --author, --after, --until, --sinc, --grep “…..”, SHA..SHA

Exemple :

Git log -4 -–oneline -–author=lavergne

### Status & diff – Visualiser les modifications

La commande status permet de faire un point sur nos modifications. Dans un premier temps s’affiche en vert toutes les modifications dans le staging index prêtes à être commit, ensuite s’affiche en rouge les liste des modifications dans le working dir. Enfin sera afficher les fichiers non-trackés (cf. commande add).

Comme sont nom l’indique la commande diff permet d’afficher des différences entre les étages de notre git local. Par défaut la commande afficher les différences entre les working dir. Et le local Repo. On utilisera l’obtion –staged pour regarder les différences entre le staging indx et le local Repo.

On peut effectuer un diff entre deux branches sous cette forme :

Git diff master..4G

### Rm et mv – Supprimer, renommer ou déplacer des fichiers

Malgré le fait que les actions supprimer ou déplacer un fichier ne soit pas une modification classique elle sont gérée comme tel par git.

La commande rm va donc supprimer un ficher de notre working dir. Mais elle va également directement ajouter la modification effectuée (la suppression si vous avez suivi) dans le staging index.

La commande mv permet de déplacer et de renommer des fichiers/dossiers « à la linux » La commande se présente sous la forme : git mv <source> <destination>   
Pour effectuer un renommage source et destination ont presque identique, on change juste le nom.

### clone – Récupérer un projet GitHub

*Nous reviendrons plus tard en détail sur l’utilisation d’un repository en ligne.*

La commande clone permet de récupérer la branche master d’un projet en ligne, elle se présente sous cette forme :

Git clone ʺ https://github.com/LavergneC/Projet.gitʺ

### Fetch & Merge – Mettre à jour depuis GitHub

*Nous reviendrons plus tard en détail sur l’utilisation des branches.*

La commande fetch va mettre à jour notre pointeur origin/master. Pour faire simple : nous allons récupérer le projet tel qu’il est sur git hub sous forme d’une nouvelle branche nommée origin/master dans notre projet. **Cette commende est 100% sûr** car elle de modifie rien sur notre travail.

Une fois la branche origin/master crée il va falloir intégrer les modifications en ligne dans notre projet. Nous allons effectuer un merge, l’idées est de fusionner les deux versions pour qu’elles deviennent compatible. Soit cela marche du premier coup (paraît que ça arrive) soit nous allons passer en état MERGING.

git merge origin/master

En cas de conflit se référer à la partie Branche – merge de ce rapport.

Et la commande git pull alors ??? Bonne question jeune ami, cette commande est pratique, elle permet de faire d’un coup le git fetch et le git merge. Elle n’est pas conseillée pour les débutants car elle occulte ce qui se passe réellement.

Git pull

Equivaut à :

Git fetch

Git merge origin/master

### Push – Mettre mes modifications sur GitHub

Avant d’effectuer un push il faut que notre working dir. soit à jour sur la version présente sur git hub de manière à ce qu’l n’y ai pas de conflit sur le GitHub (ce qui est donc heureusement impossible) si tout est à jour un simple git push fera l’affaire.

## Les branches

Les gestions des branches est un outils puissant et très important de git, cela va permettre de partir dans des versions de codes expérimentales sans risquer de perdre une version fonctionnelle. Sans plus attendre regardons les commandes associées pour en comprendre le fonctionnement.

### Branch

Cette commande sans paramètre affiche simplement la liste des branches du projet en cour et met en évidence la branche sur laquelle on se situe.

Git branch

Avec un paramètre <nom> la commande va créer une nouvelle branche avec le nom <nom> à partir du local repo. sur lequel on se situe.

Git branch <test\_qui\_va\_rater>

### Checkout

Nous allons utiliser Checkout pour naviguer de branche en branche (Comme tarzan). Lorsque l’on bascule sur une nouvelle branche le working directory est mise à jour en conséquence. Ainsi, avant d’effectuer un checkout il faut , n’avoir aucune modification en cour. On effectuera un commit ou un stash dans le cas contraire pour avoir un working dir. propre.

Exemple :

Git checkout <test\_qui\_va\_rater>

L’obtion -b va permettre de créer une nouvelle branche puis de s’y rentre directement :

Git checkout -b <test\_qui\_va\_rater>

Equivaut à :

Git branch <test\_qui\_va\_rater>

Git checkout <test\_qui\_va\_rater>

### Renommer ou supprimer une branche

Les option -d et -D permettre de supprimer une branche, on utilise le -d lorsque tout les commits de la branche on été merge ailleurs et -D dans le cas contraire.

Exemple :

Git branch -D <test\_qui\_a\_raté>

L’obtion -m de git branche permet de renommer une branche.

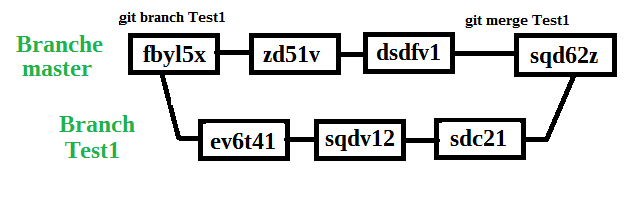
Exemple :

Git branch -m <test\_qui\_a\_raté> <Branche\_pas\_si\_mal>

### Merge

Nous ne sommes pas ici sur la commande la plus simple du chapitre mais de loin il s’agit de la plus intéressante. Merge une branche sur une autre, c’est ajouter les commit de l’une sur l’autre. Les merges les plus simple sont appelés « fast-forward » ou « ff » ils intervienne lorsque la branche mergé n’a pas de retard sur la branche sur laquelle est va arrivée. En pratique on se débrouille pour que ce scénario arrive le plus souvent possible car c’est ~~vachement~~ beaucoup plus simple.

Souvent les merge se font avec les branches non à jour l’une par rapport ç l’autre suivant cette idée (les commits sont représentés par leur SHA) :

Ainsi, le commit sqd6 contient tout les commit de la branch Test1. Pour faire ce qu’lon observe ici il faut se placer sur la branche master avant de faire la commande.

Git merge Test1

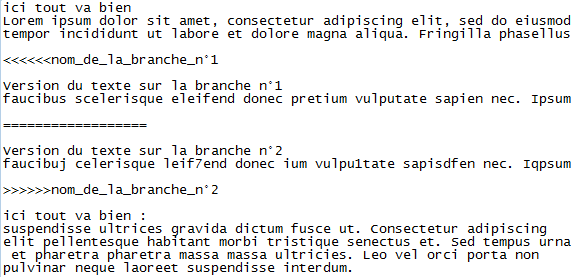
Dans le cas d’un merge Fast-forward mais aussi sur les merging très propre tout fonctionne parfaitement. D’autre fois il y a des confits. Un conflit intervient lorsque deux modifications chacune sur une branche modifier la même ligne (ou groupe de ligne) d’une façon différente. Avant de faire un conflit git va essayer de régler ça par lui-même (brave type ce git) mais ce n’est pas toujours possible. On passe alors en état (MERGING).

A cette étape git nous affiche les fichiers concernés, si tout va bien il suffit d’ouvrir les fichiers et de modifier les lignes posant un problème. A ce moment nous pouvons éprouver une peur légitime et abandonner le combat avec un :

Git merge –abort

Qui annule simplement le git merge.

Sinon il faut résoudre les conflits : cette étape est simplifiée par git qui affiche pour charque conflit les versions de chaque branche, sous cette forme :



Pour régler le confit in faut supprimer tous les <<<,>>>>, ===== générés par git et modifier le document de sort qu’il soit exactement comment on le souhait à la fin.

Dès que l’on à résolue les conflits d’un fichier on exécute la commande :

git add <fichier>

Quand tout est résolue on effectue la commande :

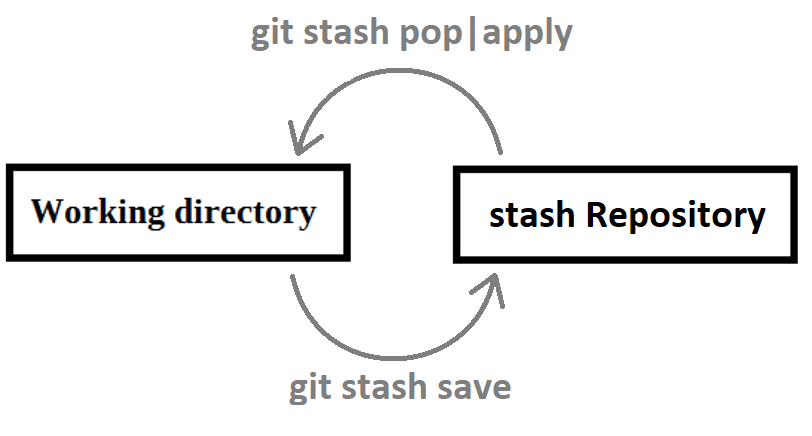
Git commit

Et la commande git pull alors ??? Cette commande est pratique, elle permet de faire d’un coup le git fetch et le git merge. Elle n’est pas conseillée pour les débutants car assez opaque.

## Stash

### Visualisation

La commande stash peut être vu comme un nouvel espace dans l’architecture de git, je me le représente sous cette forme :

Ce nouvel espace est lié à notre working directory par trois commandes que nous expliquerons un peu plus tard.

Il est important de noter que la commande checkout n’est **aucun effet** sur le stash, en effet stash peut se traduire par réserve. Ce qui signifie que l’on va pouvoir y transférer nos modifications, cela est utile dans plusieurs cas.

### Cas d’utilisations

* Je me rends compte que mes dernières modifications ne sont pas pertinentes. Je veux annuler tous les changements et revenir sur la version du local repo.. J’effectue alors un git stash save qui remet le working directory dans l’état du Local Repo.
* Les modifications effectuées sur mon Working directory sont prenantes mais je veux les commits sur une autre branche. Dans ce cas je place mes modifications dans mon stash, je checkout la branche où je veux faire les modifications, et je récupère les modifications avec un pop ou un apply, je peux ensuite les add et les commit.
* Je suis en train de travailler et je veux aller voir sur une autre branche le travaille d’un collègue, je ne peux pas checkout tant que mon working directory n’est pas propre mais je ne veux pas faire de commit maintenant car mon travail n’est pas terminer. Je fais un got stash save, je peut maintenant de déplacer de branche en branch librement. Lorsque je revient sur ma branche je récupère mon travail en cour avec un pop ou un apply.

### Les commandes stash

* Git stash save permet de transférer les changements non staged depuis le working directory vers le staging repo.. Chaque stach save va créer un « stash » regroupant tous les changements transférés.
* Git stash list permet d’afficher la liste de nos stashs avec leur ID. Le premier stash prend l’index 0 puis chaque stash suivant prend la valeur suivante, 1 puis 2, etc.
* Pour les trois commandes suivantes si le nom du stash n’est pas renseigné, l’action sera effectuer avec le dernier stash créé.
* Git stash apply [nom du stash] va **copier** les changements du stash vers le working directory. . Le stash concerné est donc **concervé** dand la liste des stashs.
* Git stash pop [nom du stash] va **transférer** les changements du stash vers le working directory. Le stash concerné est donc **supprimé** de la liste des stashs.
* Git stash drop [nom du stash] va supprimer le stash concerné
* Git stash clear supprime **tous** les stashs.