

GIT OVERVIEW

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

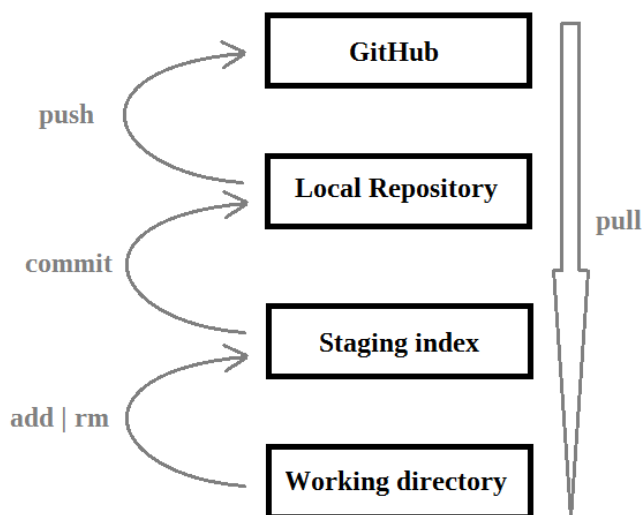
TABLE DES MATIERES

Git overview	1
Principe de git	2
Commandes les plus courantes	2
Config - Initialisation des variables locales	2
Add et commit.....	3
Log – Afficher des commits	4
Status & diff – Visualiser les modifications	4
Rm et mv – Supprimer, renommer ou déplacer des fichiers.....	5
clone – Récupérer un projet GitHub	5
Fetch & Merge – Mettre à jour depuis GitHub	6
Push – Mettre mes modifications sur GitHub.....	7
Les branches.....	7
Branch	7
Checkout	7
Renommer ou supprimer une branche	8
Merge	8
Stash.....	10

Visualisation	10
Cas d'utilisations.....	12
Les commandes stash.....	11

PRINCIPE DE GIT

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



- Le working directory correspond à notre espace physique de travail.
- Le staging index est l'espace intermédiaire entre le repository et le working index. On va pouvoir y regrouper nos modifications, pour ensuite les commit
- Le local repository représente l'image de notre travail sauvegardé par une suite de commit, sur notre machine

- GitHub ou gitLab, héberge une copie de notre local Repository en ligne.

Sous git, hormis le working directory **tout est commit**, lorsque l'on effectue un push on n'envoie pas le nouveau code mais **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 utilisées par git lors de l'exécution d'une commande, nous utilisons un paramètre :

- Soit : **--system** qui permet de modifier la variable 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 utilisées concernant le dépôt local. La commande Add permet d'ajouter les modifications effectuées sur un fichier dans 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 staging index et les envoyer dans notre repository. Un commit est un agrégat de modifications portant une description et un SHA. Le paramètre -m que l'on mettra toujours (sauf pour un merge où git s'en charge...) donne le nom à notre paquet.

A utiliser avec parcimonie : La commande **git commit -am "message"** 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 "Ajout de l'init GPIO"
```

Bonnes pratiques :

- Un commit doit porter sur un sujet en particulier
- Le message doit être clair et précis, cela rendra un grand service à vous du futur et aux personnes qui vous aideront (ces deux personnes étant sympathique)
- Le message doit être court, si votre message est trop long, vous devriez surement subdiviser votre commit.

Log – Afficher des commits

Cette commande en elle-même est très simple mais les options associées sont très riches. La commande permet d'afficher des commits. Comme un projet contient souvent beaucoup de commit on utilisera donc 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 affiché sur une seule ligne
- `--since date` : afficher des commit depuis une date donnée
- En vrac : `--author`, `--after`, `--until`, `--since`, `--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 affiché les fichiers non-trackés (cf. commande add).

Comme son nom l'indique la commande diff permet d'afficher des différences entre les étages de notre git local. Par défaut la commande affiche les différences entre le working dir. et le local Repo. On utilisera l'option **--staged** pour regarder les différences entre le staging index 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 fichier 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 "https://github.com/LavergneC/Projet.git"
```

```
git clone git@github.com:LavergneC/Git-overview.git
```

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 gitHub/gitLab sous forme d'une nouvelle branche nommée origin/master dans notre projet. **Cette commande est 100% sûr** car elle ne modifie rien sur notre travail.

Une fois la branche origin/master créée il va falloir intégrer les modifications en ligne dans notre projet. Nous allons effectuer un merge, l'idée est de fusionner les deux versions pour qu'elles deviennent compatibles. Soit cela marche du premier coup soit nous allons passer en état MERGING.

```
git merge origin/master
```

En cas de conflit se référer à la partie Branche : [Merge](#).

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 occulte ce qui se passe réellement mais fait gagner beaucoup de temps.

```
Git pull
```

Equivalut 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'il n'y ai pas de conflit sur le GitHub (ce qui est heureusement impossible) si tout est à jour un simple *git push* fera l'affaire.

LES BRANCHES

La gestion 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 [git](#) commit

À tout moment on peut effectuer la commande suivante pour savoir où on en est :

```
git status
```

Stash dans le cas contraire pour avoir un working dir. propre.

Exemple :

```
Git checkout <test_qui_va_rater>
```

L'option **-b** va permettre de créer une nouvelle branche puis de s'y rendre directement :

```
Git checkout -b test_qui_va_rater
```

Equivalait à :

```
Git branch test_qui_va_rater
```

```
Git checkout test_qui_va_rater
```

Renommer ou supprimer une branche

Les options **-d** et **-D** permettent de supprimer une branche, on utilise le **-d** lorsque tous les commits de la branche ont été merge et **-D** dans le cas contraire.

Exemple :

```
Git branch -D test_qui_a_raté
```

L'option **-m** de git branch permet de renommer une branche.

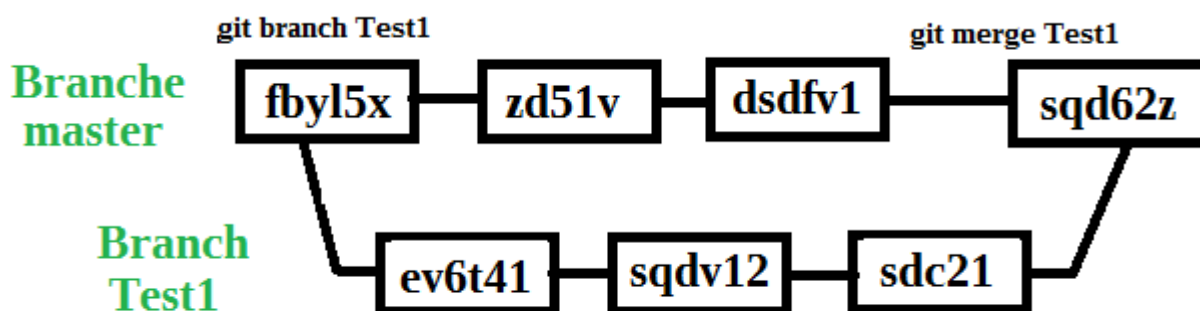
Exemple :

```
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 interviennent lorsque la branche mergée n'a pas de retard sur la branche sur laquelle est va arriver. En pratique on se débrouille pour que ce scénario arrive le plus souvent possible car c'est ~~vachement~~ plus simple à gérer.

Petit dessin paint : (les commits sont représentés par leur SHA) :



Ainsi, le commit **sqd6** contient tout les commit de la branch **Test1**. Pour faire ce qu'on 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 merge propre tout fonctionne parfaitement. D'autre fois il y a des conflits. Un conflit intervient lorsque deux modifications, chacune sur une branche, modifient 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 tout 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 chaque conflit les versions de chaque branche, sous cette forme :

```
ici tout va bien
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Fringilla phasellus

<<<<<nom_de_la_branche_n'1
version du texte sur la branche n'1
faucibus scelerisque eleifend donec pretium vulputate sapien nec. Ipsum
=====
version du texte sur la branche n'2
faucibus celerisque leifend donec ium vulputate sapisdfe nec. Iqpsum
>>>>>nom_de_la_branche_n'2

ici tout va bien :
suspendisse ultrices gravida dictum fusce ut. Consectetur adipiscing
elit pellentesque habitant morbi tristique senectus et. Sed tempus urna
et pharetra pharetra massa massa ultricies. Leo vel orci porta non
pulvinar neque laoreet suspendisse interdum.
```

Pour régler le conflit il faut supprimer tous les <<<,>>>, ===== générés par git et modifier le document de sorte qu'il soit exactement comment on le souhaite à la fin.

Dès que l'on a résolu les conflits d'un fichier on exécute la commande :

git add <fichier>

Quand tout est résolu on effectue la commande :

git commit

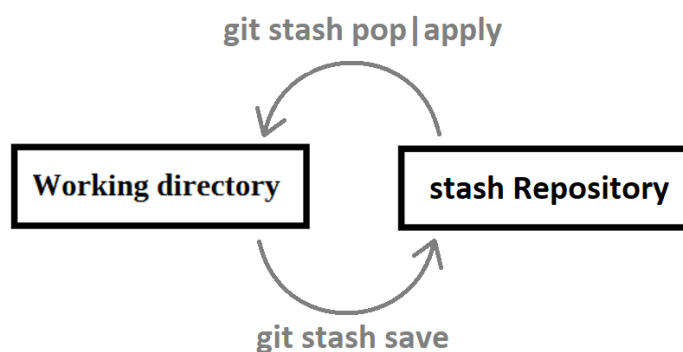
À tout moment on peut effectuer la commande suivante pour savoir où on en est :

`git status`

STASH

Visualisation

La commande stash peut être vue 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.

Les commandes stash

- `Git stash save` permet de transférer les changements non staged depuis le working directory vers le staging repo.. Chaque stash save va créer un « stash » regroupant tous les changements transférés. Le « save » est facultatif.
- `Git stash list` permet d'afficher la liste de nos stashes avec leur ID. Le dernier stash prend l'index 0 puis chaque stash précédent 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 effectuée 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 **concerné** dans la liste des stashes.
- `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 stashes.
- `Git stash drop [nom du stash]` va supprimer le stash concerné de la liste
- `Git stash clear` supprime **tous** les stashes.


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 correctes 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 de travail 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 terminé. Je fais un `git stash save`, je peux maintenant de déplacer de branche en branche librement. Lorsque je reviens sur ma branche je récupère mon travail en cours avec un `pop` ou un `apply`.

ANNULER DES CHANGEMENTS

Parce que le principe même de git est de sauvegarder l'historique des changements que l'on effectue, annuler ces changements n'est pas toujours facile, les commandes prévues à cet effet sont assez spécifiques, elles dépendent beaucoup du contexte.

1. Dans le working directory : ***git checkout -- <fichier>***

 Attention aux **espaces**

2. Si on souhaite annuler un changement dans le staging index : (cette commande laisse la modification dans le working directory)

git reset HEAD fichier

3. Annuler le dernier commit : ***git commit --amend -m "monMessage"***

- Il faut add avant, la commande permet donc de remplacer le dernier commit
- Si nous n'avons pas add avant, cette commande ne fait que changer le message du dernier commit