TRAVAIL EN GROUPE - GIT

Soit une version **A** d'un projet.

2 participants "tirent" (*git pull*) la version **A**.

- Si la 1ere modifie **A** pour devenir une version **B** et qu'il la "commit", **A** n'est plus la version en vigueur.
- Si la 2e modifie **A** pour devenir une version **C** et qu'il essaie de la commit, il va y avoir un conflit, car c'est la version **B** qui est en vigueur, cela génère une erreur.

Il faut alors récupérer **B** et vérifier les différences. Si **C** n'impacte pas les modifications apportées par **B**, on devrait pouvoir merger **B** et **C** pour donner une version **D** qui deviendra master une fois **D** commit.

QUELQUES COMMANDES DE BASES

Transformer un répertoire en dépôt: git init. Un dossier .git est généré.

git status nous indique où on en est (branche, untracked files etc.)

git add permet de tracker les fichiers.

git commit permet la création d'un historique des modifications accompagnées d'un message personnalisé.

git log donne accès à l'historique.

git diff permet de visualiser les différences entre 2 fichiers (plus ? peut être ?)

Soit on git add pour accepter les différences,

Soit on git checkout -- pour remettre le fichier dans son état initial.

LES BRANCHES

https://www.youtube.com/watch?v=0x6TvDQYu7Y&list=PLVQYiy6xNUxxhvwi0PGmXb5isUdVwmsg8&index=18

git branch pour voir la branche sur laquelle on est, et pour voir toutes les branches aussi

git branch <nvbrnch> crée une branche.

git checkout <brnch> permet de se positionner sur la branche <brnch>

git checkout <sha_commit> permet de se positionner sur un commit en particulier dont on fournit le sha en paramètre (on obtient ce sha en naviguant dans l'historique avec git log). Il y aura un message qui s'affichera pour dire que ce n'est pas très legit comme méthode et qu'il vaut mieux passer par un checkout -b <brnch> d'abord.

git checkout -b <brnch> crée la branche si inexistante et se positionne dessus.

On peut démarrer une branche depuis un commit:

git checkout <sha_commit> + git branch <brnch>

git branch -d <brnch> suppression de la branche <brnch>

git checkout -- <fichier> permet de revenir à la version précédente de <fichier> -> Cela implique que l'on perde les modifications apportées à <fichier>. Ce n'est possible que si le fichier n'est pas tracké par git add.

git reset HEAD <file> On "unstage" le fichier <file> (par exemple, celui auquel on est revenu via git checkout -- <file>). On peut alors git checkout -- fichier pour revenir à l'état précédent d'avant le add.

Pour changer le nom d'une branche:

- création d'une nouvelle branche
- suppression de l'ancienne

git log <brnch> historique des commits de la branche.

MERGE - REBASE

https://www.youtube.com/watch?v=3CDMP58P7uM&list=PLVQYiy6xNUxxhvwi0PGmXb5isUdVwmsg8&index=19&t=214s

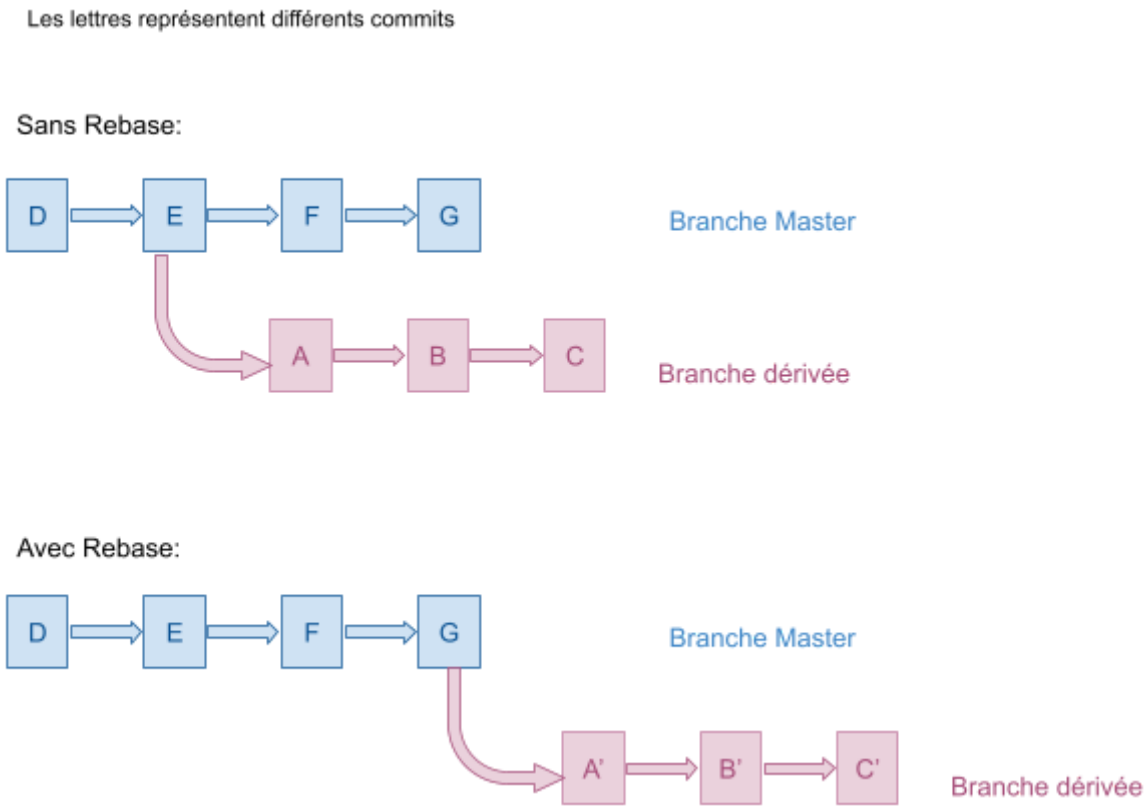
rebase : mise à jour de branches locales.

On rapatrie sur la branche **dérivée** les commits de **master** (on peut faire l'inverse mais en général, c'est dans ce sens qu'on procède).

Depuis la branche dérivée: (git checkout <dérivée> ou git branch pour vérifier la branche)

git rebase master

git rebase master dérivée



Pour envoyer une branche dérivée vers la principale en gardant un historique de master, avant de l'envoyer sur master -> git rebase

On déplace le commit en commun vers le haut (HEAD) de Master et on adjoint les commits dérivés. Les branches sont toujours séparées à ce stade.

Il peut y avoir des conflits de commit en faisant git rebase master. Il est possible qu'on ait modifié le même fichier sur les deux branches, le commit du master ayant été fait après le commit le commit de la branche dérivée, sur les mêmes lignes.

On est alors au milieu du rebase et git status nous le confirmera.

git rebase --skip annulera le commit en trop

git rebase --abort annule le rebase en cours, remet les commits sur le commit commun d'origine le temps de regler le probleme. On pourra reprendre le rebase avec git rebase --continue

git indiquera dans le fichier en conflit les différences empêchant le rebase, il suffit d'entrer dans le fichier et de choisir quelle modification garder puis on le track avec git add -> cela aura pour effet de générer un nouveau commit et de supprimer les commits en conflit. On a ainsi l'historique de commits de master dans la branche dérivée.

git commit --amend: permet de rentrer dans le dernier commit pour le modifier (et attention car cela peut être dangereux de procéder de la sorte). Si la commande bug on peut faire : git commit --amend -m "message de commit"

git rebase --help

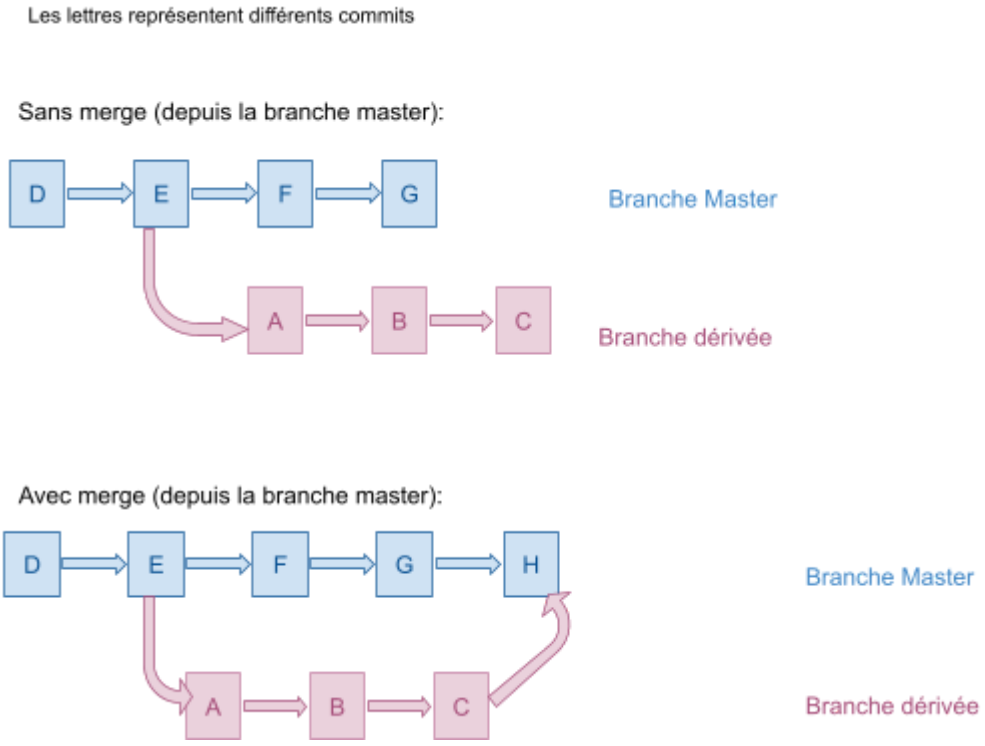
merge : rassembler 2 branches divergentes.

Le merge est l'inverse du rebase: on rapatrie sur la branche master les commits de la branche dérivée en haut de la branche master (alors que rebase on rapatrie sur la branche dérivée les commits de master).

On rapatrie sur la branche **master** les commits de la branche **dérivée** en haut de la branche master. Fusion des branches ?

Depuis la branche master :

git merge dérivée



D'après le manuel:

Then git merge <derivee> will replay the changes made on the derivee branch, since it diverged from master until its current commit, and record the result in a new commit along with the name of the two parent commits and a log message from the user describing the changes.

git merge <branche dérivée>

Si survient un conflit sur le merge (dans le même genre que vu ci-dessus avec rebase):

git merge --abort pour abandonner le merge

sinon git add + git commit pour prendre une décision -> on modifie le fichier en conflit avant le tracking. On git commit pour conclure le merge.

git merge --help

PUSH - FETCH - PULL

https://www.youtube.com/watch?v=gy3PPotpfGI&list=PLVQYiy6xNUxxhvwi0PGmXb5isUdVwmsg8&index=21&t=372s

git remote add nom url : ajouter un remote au repository, nom = origin c'est une sorte de convention.

git push: au 1er push si aucun args (origin master en l'occurrence), un message d'erreur apparaîtra, car la branche locale n'est associée à aucune branche distante (upstream), git propose alors la commande suivante : git push --set-upstream origin master, sinon on peut tout à fait utiliser la commande standard de push : git push origin master.

Si on fait un git branch juste après on a deux branches: *master et remotes/origin/master -> c'est une nouvelle branche sur le remote. On a alors cette branche aussi en locale, c'est à dire que si on se mettait sur une nouvelle machine, et qu'on git clone le repository, on se retrouverait avec ces-dites branches: git aurait appliqué tout seul l'option --set-upstream et on pourrait simplement appliquer git push.

git fetch est une commande qui permet de récupérer sur la machine A, les modifs apportées à une branche et pushées depuis une machine B.

git fetch --all récupère les modifications de toutes les branches du repo. Si on a fait une modif en local sur la branche master, on pourra rapatrier origin/master avec un git rebase origin master

Il existe une commande qui nous applique directement git fetch && [git rebase || git merge], il s'agit de git pull.

Si on fait git pull, alors c'est fetch et merge qui s'applique, sinon si on veut rebase plutôt que merge, on fera git pull --rebase. En général, c'est plutôt cette dernière qu'il faut utiliser car on sera plutôt à travailler sur une branche dérivée et il faudra y faire apparaître les commits de la branche master.

Le fetch se fera sans soucis mais il peut y avoir des conflits sur le rebase si la branche locale n'est associée à aucune branche du remote. on aura recours à la commande suivante pour résoudre le conflit:

git branch --set-upstream-to=origin|<branch> master

ou

git pull --rebase origin master

Pour se passer de l'option --set-upstream, lors du 1er push on peut faire

git push -u origin master

C'est une façon de lui dire "tu pushes sur cette adresse et tu vas faire en sorte que ma branche locale suive ma branche distante". Il nous sera retourné "Branch master set up to track remote branch master from origin" si tout est ok.