Git v2.0 19/12/2015

##
LEGENDE:
##
Les éléments entre [ ] sont des options non obligatoires
Ex: [REF] -> Il s'agit du référence optionnelle
Les éléments en MAJUSCULE sont à remplacer par l'utilisateur
Ex: BRANCH -> Il faut nommer la branche
Note: Seuls de rare élément en MAJUSCULES sont à reprendre tel quel. Comme HEAD ou -D, -C etc…
##

Installer Git sur MAC:
$ brew install git
//ou
https://git-scm.com/downloads

Installer Git sur WINDOWS:
https://git-for-windows.github.io/<script
//Permet d'installer un terminal particulier où les commandes Git fonctionneront sous l'OS (GitBash)

Vérifier la version de Git:
$ git --version

3 niveaux de config de Git:
1) Système : /etc/gitconfig
//La config impactera tous les utilisateurs du système
2) Global : ~/.gitconfig
//La config impactera l'utilisateur courant
3) Local : .git/config
//La config impactera uniquement le dépôt dans lequel le dépôt Git a été initialisé

Afficher les 3 config:
$ git config -l
//ou
$ git config --list

Ajouter un élément dans la config:
$ git config [--global|--system] KEY VALUE
//On précise le niveau de config, global ou system. Si rien n'est mis, ça sera enregistré sur la config locale
$ git config [--global] -e
//Pour ouvrir le fichier directement et pouvoir modifier la config à la main

Configuration minimale :
$ git config –global user.name NAME
$ git config –global user.email EMAIL
//Configuration du nom d'utilisateur et de l'email (OBLIGATOIRE POUR UTILISER GIT)

Quelques éléments de config :
$ git config --global color.ui true
//Pour activer la couleur dans git
$ git config --global help.autocorrect 1
//Activation de l'autocorrection
$ git config --global core.autocrlf true
//à Faire sous windows

Git help :
$ git help nomdelacommande
//Affiche l'aide pour la commande donnée
$ git help nomdelacommande -w
//Pour que l'aide s'ouvre sur le navigateur

Créer des ALIAS:
$ git config --global alias.ALIAS CMD
Exemple: $ git config --global alias.st status
//Alias st pour la commande status
$ git config -l
//Affiche la config (dont tous les alias)

Gestion du repository Git:
$ git init
//Génère le repository Git, à faire dans le dossier ou est le projet déjà existant ou dans le dossier nouvellement créé
$ git clone URL [DIR]
//Récupère un projet existant (sur Github par exemple). Il faut spécifier l'URL et le nom du dossier (DIR) ou est le repository
$ git log --pretty=oneline --abbrev-commit --graph --decorate --all
//Afficher historique d'un repository Git (plus joliment)
$ git log [REF]
//Afficher historique d'un repository Git
$ git l
//Afficher historique d'un repository Git (plus graphique)
$ git log -n 2
//Afficher les deux derniers COMMIT
$ git log --oneline
//Afficher l'historique en format condensé
$ git log -n 2
//Afficher les deux derniers COMMIT
$ git log -p FILE
//Affiche les COMMI en rapport avec le FILE cité
$ git checkout [REF]
Le pointeur <HEAD> représente l'état de la copie locale
//Permet de naviguer dans l'historique en remettant le projet dans l'état dans lequel il était à ce commit REF (On peut voir l'état du projet à un commit donné)

Les 4 états de fichier dans Git:
1) UNMODIFIED (Non modifié)
2) UNSTAGED (Modifié mais non indexé)
3) STAGED (Modifié et indexé)
4) COMMITED (Commité)

$ git status
//Afficher le status du repository

Modifier l'état des fichiers:
$ git add FILE
//Ajoute d'un fichier dans la STAGING AREA
$ git add --all //OU// git add -A
//Ajoute toutes les modifications dans le STAGING AREA

$ git add "*.html"
//Ajoute tous les fichiers HTML modifiés
$ git add -u *//OU//* git add --update
//Ajoute tous les fichiers modifiés et supprimés par Git
$ git add --ignore-removal .
//Ajoute tous les fichiers nouveaux et modifiés (mais pas ceux supprimés) par Git
$ git rm FILE
//Supprime le fichier de l'historique de Git
$ git mv
//Modifier le nom d'un fichier en conservant l'historique
$ git reset
// Vide la STAGING AREA
$ git reset HEAD FILE
//Remet le fichier (de l'état COMMITED) à l'état STAGED
//(Annule les changements de la STAGING AREA)
$ git reset HEAD^
//Reviens en arrière d'1 COMMIT
$ git reset HEAD^ --soft
//Reviens en arrière d'1 COMMIT mais laisse les modifications dans la STAGING AREA
$ git checkout --FILE
//Remet le fichier (de l'état STAGED) à l'état UNSTAGED
//(Annule les changements de la copie LOCALE)
$ git checkout HEAD FILE
// Combine en une commande les 2 précédents retours en arrière
$ git commit -m "message du commit"
// Commit les fichiers dans de la STAGING AREA
$ git commit -am
// Les fichiers sont STAGED et COMMITED en une fois.
// Fait le git add -u et le git commit en même temps
// le m permet de laisser un message de commit en même temps

Visualiser les changements:
$ git diff
//Afficher les changements dans la copie LOCALE
$ git diff --staged
//Afficher les changements dans la STAGING AREA

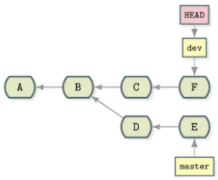
Pour utiliser le ssh:

- SI la clé est déjà générée, elle est dans ce répertoire (/Users/<votreNom>/.ssh/id_rsa)
- Sinon, pour la générer:
  $ ssh-keygen -t rsa -b 4096 -C "MAIL"

Une passphrase est demandée
Il faut ensuite aller dans : /Users/<votreNom>/.ssh et copier coller le contenu de la clé ssh qui finit par .pub (pour .public) et ajouter cette clé dans Github/Bitbucket…
Pour vérifier le bon fonctionnement du ssh, se rendre dans le dépôt et faire la commande:
$ git remote show origin

Ajouter la clé à l'agent ssh (si besoin):
$ eval "$(ssh-agent -s)"
//Vérifie que l'agent ssh est activé
$ ssh-add ~/.ssh/id_rsa
//Ajoute la clé ssh à l'agent ssh
$ pbcopy < ~/.ssh/id_rsa.pub
//Copie la clé ssh qui est dans le fichier id_rsa du dossier .ssh

Gestion de branches:

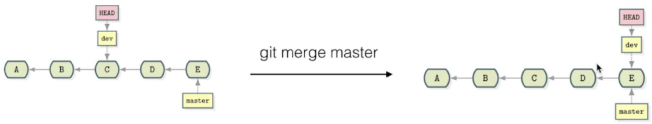


Une branche est un pointeur sur un commit

$ git branch -a
//Liste toutes les branches (Locales et distantes)
$ git branch -r
//Liste les branches Remote (Distantes)
$ git branch
//Liste les branches locales
$ git checkout BRANCH
//Permet de changer de branche
// Le pointeur HEAD se met sur la branche donnée
$ git branch BRANCH [REF]
//Permet de créer une branche et éventuellement mettre une référence, le //pointeur de la branche se posera sur le commit donné en REF
$ git checkout –b BRANCH [REF]
//Permet de créer une branche et éventuellement mettre une référence, le //pointeur de la branche se posera sur le commit donné en REF + Le //pointeur HEAD se pose sur cette nouvelle branche créée pour y travailler
$ git branch –d BRANCH
//Supprime la branche donné
//Supprimer une branche ne supprime que l'étiquette de cette branche pas les commit de la branche
$ git branch –D BRANCH
//Supprime la branche donné
//Force la suppression pour une branche non mergé

DETACHED HEAD :
Si un checkout est fait sur un commit et non sur une branche/Etiquette.
Si ensuite un commit est fait, il sera orphelin et Git supprime ces derniers au bout d'un moment.
$ git branch BRANCH COMMIT
//Crée une branche sur le commit donné pour qu'il ne soit plus orphelin

MERGE:
Se mettre sur la branche vers lequel on veut merger
$ git merge BRANCH
//Merge la branche donnée sur la branche actuelle
Si les deux références/branches à merger sont sur le même niveau (elles ne sont pas séparées). On peut faire un MERGE FAST FORWARD.



//Cela déplace le pointeur HEAD et la branche actuelle vers la branche donnée

REBASE:
$ git rebase BRANCH
//Idem MERGE mais "envoie" la BRANCH actuelle dans la BRANCH citée

Si conflit lors du MERGE:
Méthode résolution de conflits manuelle :
$ git diff
//Voir les conflits
- RÉSOUDRE les conflits en gardant/supprimant les parties désirées
- STAGER les fichiers modifiés
- Faire un COMMIT de MERGE
Méthode résolution de conflits "automatique" :
- Ouvrir le mergetool
- Résoudre les conflits
- Sauvegarder la résolution (cela commit merge automatiquement)

REMOTE:
Une remote est une adresse d'un dépôt distant
Par défaut lors d'un clone, la remote est ORIGIN
$ git remote -v
//Affiche les remote
$ git remote show REMOTE
// Affiche les infos de la remote
$ git remote add REMOTE [URL]
// Ajoute une remote
$ git remote rm REMOTE [URL]
// Supprime une remote

REMOTE BRANCH:
$ git fetch REMOTE
// Synchronise la branche d'une remote
$ git fetch --all
// Synchronise les branches de toutes les remotes
$ git branch -vv
// Affiche les correspondances Branche locale ←→ Branche distante

PULL:
$ git pull REMOTE BRANCH
ou version simple (qui nécessite les informations de tracking)
$ git pull
// Un PULL correspond à un FETCH + MERGE

PUSH:
$ git push REMOTE LOCALBRANCH:REMOTEBRANCH
ou version simple (qui nécessite les informations de tracking)
$ git push
// Un PUSH met à jour le repo distant
$ git push -f
// Force le PUSH

Travailler avec les REMOTE BRANCH:
$ git checkout -t REMOTEBRANCH
//Utiliser une REMOTE BRANCH pour la première fois
$ git checkout -b BRANCH REMOTEBRANCH
// Initialiser les informations de tracking à la création
$ git push -u REMOTE BRANCH
//PUSH d'une nouvelle branche
$ git push REMOTE --delete BRANCH
// Supprimer une branche distante

Modifier son historique:
$ git commit --amend
// Ajoute le contenu de la STAGING AREA au précédent COMMIT **[Ne pas faire sur un commit déjà PUSH]**
$ git revert REF
// Applique le patch inverse du COMMIT **[Peut être fait sur un commit déjà PUSH]**
$ git reset --hard REF
// Déplacer le pointeur de BRANCHE
// Les COMMIT fait après le COMMIT cité en REF seront orphelins
// Lors de l'utilisation, le contenu de la copie LOCAL et de la STAGING AREA sont SUPPRIMÉ!!

Le STASH:
$ git stash [save ["message"]]
Ex: $ git stash save "WIP: Add function"
// Le STASH est commun à toutes les branches et fonctionne comment une liste
$ git stash list
// Liste les différents STASH
$ git show stash@{N}
// Pour afficher un STASH précis, le "N" correspond au numéro du STASH
$ git stash pop [--index] [stash@{N}]
// Réapplique une modification (Remet les modifications déplacées dans STASH dans la copie LOCALE)
// Le POP laisse une copie de la modif dans STASH
$ git stash apply [--index] [stash@{N}]
// Réapplique une modification (Remet les modifications déplacées dans STASH dans la copie LOCALE)
// Le APPLY supprime la modif dans STASH
NOTE: Pour que le contenu de la STAGING AREA soit également récupérée, il faut préciser l'option --index

Les TAGS:
Il y a deux types de TAG:
1) TAG lourd -> Contient l'auteur, la date de création et un sha1 d'identification.
2) TAG Léger -> Est juste un marqueur d'identification

$ git tag TAG [REF]
// Créer un TAG léger
$ git tag -a [-m "MESSAGE"] TAG [REF]
// Créer un TAG lourd
$ git tag
// Liste les TAGS existants
$ git checkout [-b NEWBRANCH] TAG
// Utiliser un tag
$ git push --tags
*// Partager tous les TAGS (***Le PUSH ne partage pas les TAGS par défaut***)*
$ git push REMOTE TAG
*// Partager un tag (***Le PUSH ne partage pas les TAGS par défaut***)*
$ git tag -d TAG
// Supprime le TAG **LOCAL**
$ git push --delete REMOTE TAG
// Supprime le TAG **DISTANT**

Le Gitignore:
// Liste les éléments (fichiers, dossiers…) que Git ne devra pas partager dans le projet.
Ex: https://github.com/github/gitignore

Le REFLOG:
$ git reflog
// Pour retrouver les COMMITS orphelins

Le CHERRY-PICK:
$ git cherry-pick REF
// La commande CHERRY-PICK permet de déplacer un ou plusieurs COMMIT d'une BRANCH à une autre.
// Rajoute le commit à la BRANCH ACTUELLE
Si il y a un ou plusieurs conflits, les régler.
Puis entrer la commande suivante:
$ git cherry-pick --continue
// Permet de continuer le cherry-pick (vérifie si il y a d'autres conflits)
// Si oui, les régler et re taper la commande
// Si non, le cherry-pick est terminé

Un outil sympa UNGIT:
$ npm install -g ungit
// Installer UNGIT
$ ungit
//Lancer UNGIT