GIT: Utilisation de Git en ligne de commandes

Table of Contents

Il n’y a rien que les outils graphiques ne fassent que l’interface en ligne de commande ne puisse faire ; la ligne de commande reste l’interface qui donne le plus de puissance et de contrôle sur les dépôts.

Mettre à jour son dépôt local

```
$ git pull
```

ou

```
$ git pull ssh://username@host/var/cache/git/myproject/
```

Ajouter des fichiers

```
$ git add *
```

Commiter les changements

Pour mettre en même temps un log et commiter, le tout en une seule ligne (sinon enlever -m...) :

```
$ git commit -a -m "Mon premier ajout"
```

Ceci va donc mettre à jour en local le repository (ne pas oublier le push ensuite si sur le serveur distant).

Changer l'éditeur et le visualiseur par défaut

Pour forcer par exemple vim et less :

```
$ git config --global core.editor vim
$ git config --global core.pager "less -FSRX"
```

Obtenir les logs

Cnsulter à n'importe quel moment les logs des anciens commits via cete commande :

```
$ git log
```

Pour obtenir les logs d'un fichier bien précis :

```
$git log test
```

Si le fichier a été renommé, utiliser l'option –follow pour voir également son nouveau nom.

Pour voir les logs des 2 derniers jours :

```
$ git log --since="2 days ago"
```

Faire une recherche

Il est possible de faire une recherche de contenu dans les fichiers comme ceci :

```
$ git grep deimos *
```

Ignorer des fichiers

Pour ne pas autoriser certains fichiers tel que les fichiers temporaire de vim par exemple. Il va falloir placer un fichier .gitignore qui sera à la racine de la copie de travail qui sera en général ajouté lui même au dépôt, soit dans le fichier .git/info/exclude. Voici un exemple :

```
Configuration File .gitignore
*~
*.o
```

S'assurer qu'il n'y a rien à commiter

Il est possible de s'assurer qu'il n'y a rien a commiter de cette façon :

```
$ git stash
```

```
  No local changes to save
```

Configurer des alias

Il est possible de configurer des alias pour les commandes un peu longues. Par exemple :

```
$ git config --global alias.co 'checkout'
$ git config --global alias.ci 'commit -a -m'
```

Ici 'git co' correpond à la commande 'git checkout'. Cette configuration (si elle est globale ou non) peut être située à 2 endroits :

- ~/.gitconfig pour en bénéficier dans tous les dépôts.
- .git/config d’un projet pour restreindre son accès à cet unique projet.

Revert : annuler un commit

Git revert permet d'annuler le précédent commit en en introduisant un nouveau.

Tout d'abord faire un git log pour récupérer le numéro du commit :

```
$ git log
```

```
  commit : bfbfefa4d9116beff394ad29e2721b1ab6118df0
```

Puis faire un revert pour annuler l'ancien :

```
$ git revert bfbfefa4d9116beff394ad29e2721b1ab6118df0
```

En cas de conflit, il suffit de faire les modifications à la main, de refaire un add puis un commit.

Note : on peut aussi utiliser les quelques premiers caractère de l'identifiant (SHA-1) du commit tel que bfbfe. Git fera automatiquement la completion (a la condition qu'un autre commit ne commence pas avec les mêmes caractères).

Reset : effacer tous les commits à partir d'un ancien

Git reset va supprimer complètement tous les commits effectués après une certaine version :

```
$ git reset bfbfefa4d9116beff394ad29e2721b1ab6118df0
```

Git reset permet de réécrire l'histoire. Celà peut avoir de grave conséquences si on travaille à plusieurs sur un projet. Non seulement parce que les repository en locale des autres personnes vont devoir être resynchroniser, mais en plus parce qu'il peut y avoir beaucoup de conflits. Il est donc conseiller d'utiliser cette commande avec précaution.

Ensuite il sera nécessaire de refaire un 'git add' des fichiers, puis un 'git commit'.

Si on veut que la copie de travail en local reflète le dépôt, ajouter l'option –hard lors du reset :

```
$ git reset --hard bfbfefa4d9116beff394ad29e2721b1ab6118df0
```

En cas d'erreur pour revenir en arrire, git a conservé l'état précédent, donc :

```
$git reset --hard ORIG_HEAD
```

HEAD signifie le dernier commit par défaut.

En cas de message du type :

```
! [rejected]        master -> master (non-fast forward)
```

Pour vraiment forcer le commit :

```
$ git push origin +master
```

Restaurer un fichier d'un ancien commit

Pour restaurer uniquement un fichier bien particulier d'un ancien commit via la commande checkout :

```
$ git checkout bfbfe test
```

Supprimer un fichier

Il existe 2 méthodes :

```
$ git rm mon_fichier
```

La 2ème méthode consite à faire un rm standart, puis un git commit -a, il détectera que ce fichier n'existe plus et le supprimera du dépôt.

Déplacement d'un fichier

Pour déplacer un fichier au sain du dépôt git, il faut utiliser cette commande :

```
$ git mv test1 test2
```

Lister le contenu du dépôt

On peut utiliser cette commande pour lister le contenu d'un dépôt :

```
$ git ls-files
```

Note: les dossiers vide ne seront pas afficher

Créer une archive du dépôt

Pour faire une archive au format tar gzip, nous allons utiliser l'option archive :

```
$ git archive --format=tar --prefix=version_1/ HEAD | gzip >
../version_1.tgz
```

- –prefix : permet de donner un dossier qui sera créer lors de la décompression.
- HEAD : permet de spécifier le commit (ici le 1er)

Voir un fichier en particulier d'un commit

Il est possible de voir un fichier (README) d'un commit :

```
$ git show 291d2ee31feb4a318f77201dea941374aae279a5:README
```

ou voir tous les diffs d'un coup si on ne spécifie pas le fichier :

```
$ git show 291d2ee31feb4a318f77201dea941374aae279a5
```

Réécrire l'histoire

Pour revenir en arrière, il faut faire un rebase interactif en indiquant le commit le plus loin que l'on souhaite modifier:

```
$ git rebase -i cbde26ad^
```

Dans l'éditeur, modifier 'pick' en 'edit' sur la ou les lignes à modifier. Enregistrer et quittez. Faire tous les changements et ensuite valider le commit:

```
$ git commit -a --amend
```

ou

```
$ git commit -a --amend --no-edit
```

Ensuite, pour passer au commit suivant:

```
$ git rebase --continue
```

Jusqu'à ce que ce soit terminé.

Faire des diff entre différentes versions

Il est très facile avec git de faire des diff entre différentes versions. Par exemple, de la version HEAD à 2 versions précédentes :

```
$ git diff HEAD~2 HEAD
```

Modifier le texte du dernier commit

Pour voir l'état du dernier log, faire :

```
$ git log -n 1
```

Pour modifier le texte de ce dernier log, utiliser l'option amend :

```
$ git commit -a --amend
```

Cela permet donc de changer juste le texte, il n'y aura pas de nouveau numéro commit. Seul le numéro d'identification changera.

Modifier la date du dernier commit

Il est possible de modifier la date du dernier commit:

```
GIT_COMMITTER_DATE="`date`" git commit --amend --date "`date`"
```

Ou sinon on peut spécifier la date en dur:

```
GIT_COMMITTER_DATE="Fri Nov 17 12:00:00 CET 2014" git commit --amend --date
"Fri Nov 17 12:00:00 CET 2014"
```

Modifier l'auteur de commits

Lorsque l'on utilise plusieurs comptes Git sur une même machine, il arrive facilement de se tromper lorsque l'on clone un repository et que l'on ne met pas le bon username et adresse mail. Du coup lorsque l'on s'en rend compte, il est trop tard et il faut réécrire une partie d l'histoire pour remettre les bonnes informations:

```
$ git filter-branch --env-filter '
```

```
  oldname="old username"
  oldemail="old email address"
  newname="new username"
  newemail="old username address"
  [ "$GIT_AUTHOR_EMAIL"="$oldemail" ] && GIT_AUTHOR_EMAIL="$newemail"
  [ "$GIT_COMMITTER_EMAIL"="$oldemail" ] && GIT_COMMITTER_EMAIL="$newemail"
  [ "$GIT_AUTHOR_NAME"="$oldname" ] && GIT_AUTHOR_NAME="$newname"
  [ "$GIT_COMMITTER_NAME"="$oldname" ] && GIT_COMMITTER_NAME="$newname"
  ' HEAD
```

Synchroniser une partie d'un dépôt

Pour n'avoir qu'une partie d'un dépôt (également appelé sparse), récupérer le dépôt dans son intégralité, puis lui spécifier que ce qu'il faut garder:

```
$ git clone ssh://deimos@git/var/cache/git/git_deimosfr .
$ git config core.sparsecheckout true
$ echo configs/puppet/ > .git/info/sparse-checkout
$ git read-tree -m -u HEAD
```

Ici dans le dépôt git_deimosfr, seul le dossier “configs/puppet/” est conservé. On peut ajouter plusieurs dossiers dans le fichier “.git/info/sparse-checkout” ligne par ligne pour conserver plusieurs fichiers.

Utiliser un repository git externe dans un git

Pour intégrer un autre Git dans un dépôt existant, il vous faudra utiliser les submodules. Par exemple pour faire pointer certains modules du dossier local puppet vers les git externe:

Ajouter la source externe à ce git :

```
$ git submodule add git://git.black.co.at/module-common
configs/puppet/modules/common
```

```
  Cloning into configs/puppet/modules/common...
  remote: Counting objects: 423, done.
  remote: Compressing objects: 100% (298/298), done.
  remote: Total 423 (delta 155), reused 203 (delta 70)
  Receiving objects: 100% (423/423), 53.96 KiB, done.
  Resolving deltas: 100% (155/155), done.
```

Commiter et push pour rendre cette configuration valable sur le serveur.

Si un client qui fait un pull. Il va récupérer toute l'arborescence, mais pas les liens externes, pour cela, il devra exécuter les commandes suivantes :

```
$ git submodule init
```

```
  Submodule 'configs/puppet/modules/common' (git://git.black.co.at/module-
common) registered for path   'configs/puppet/modules/common'
  > git submodule update
  Cloning into configs/puppet/modules/common...
  remote: Counting objects: 423, done.
  remote: Compressing objects: 100% (298/298), done.
  remote: Total 423 (delta 155), reused 203 (delta 70)
  Receiving objects: 100% (423/423), 53.96 KiB, done.
  Resolving deltas: 100% (155/155), done.
  Submodule path 'configs/puppet/modules/common': checked out
'ba28f3004d402c250ef3099f95a1ae13740b009f'
```

Si lors d'un clone, vous souhaitez que les submodules soient également prit avec, vous devez rajouter l'option “–recursive”. Exemple :

```
$ git clone --recursive git://git.deimos.fr/git/git_deimosfr
```

Les branches

Création d'une branche

Si par exemple, la version actuelle que vous avez sur git vous convient et que vous souhaitez la passer en stable, il vous faut utiliser les branches et alors créer une nouvelle branche en tant que branche de développement :

```
$ git branch devel
```

Envoyer une branche au serveur

Pour envoyer une branche au serveur :

```
$ git push origin <ma_branche>
```

Lister des branches

Maintenant, la branche devel est créer, pour le vérifier :

```
$ git branch -a
```

```
    devel
  * master
```

L''*' signifie la branche courante (en cours d'utilisation).

Lorsque vous utilisez des branches distantes, vous pouvez les lister via cette option :

```
$ git branch -r
```

```
    devel
  * master
```

Changement de branche

Pour changer de branche, il suffit d'utiliser l'option checkout :

```
$ git checkout devel
```

```
  Switched to branch "devel"
```

Vérifier :

```
$ git branch
```

- devel

```
  master
```

Pour repasser sur la branche précédente:

```
$ git checkout -f master
$ git branch
```

- master

```
  devel
```

L'option -f correspond à l'option –hard du reset qui permet de synchroniser en local les changements effectués sur le dépôt.

Pour travailler sur une branche distante :

```
$ git checkout deimos/myproject -b master
$ git branch
```

- master

```
  devel
```

Récupérer une branche depuis un serveur distant

Une fois que votre “pull” est fait, changez de branche comme ceci (vous pouvez la nommer différemment si vous le souhaitez) : Command git

```
$ git checkout -b <new_branch> origin/<new_remote_branch>
```

- **<new_branch>**: le nom de ma nouvelle branche locale
- **<new*remote*branch>**: le nom de la nouvelle branche distante (côté serveur)

Supprimer une branche

Pour supprimer une branche, rien de plus simple :

```
$ git branch -d devel
```

Si celà fonctionne parfait, si vous souhaitez forcer en cas de problème, utiliser '-D' à la place de '-d'. Pour appliquer les changements sur le serveur remote : Command git

```
$ git push origin :devel
```

Ceci supprimera la branche devel sur le serveur distant.

Merger des branches

L'option merge consiste à fusionner 2 branches complètement. Comme par exemple faire fusionner une branche devel avec une stable pour que la devel devienne stable :

```
$ git checkout master
$ git merge devel
```

Rebase : Appliquer un patch sur plusieurs branches

Par exemple si bug en master a été découvert, il est normal que si le master bénéficie du fix, la branche devel également. Se placer dans la branche master, appliquer le patch, puis merger avec la branche devel le patch. Se placer sur la branche devel, puis exécuter ces commandes :

```
$ git checkout devel
$ git rebase master
```

Il est possible de le faire de façon interactive grâce à l’argument -i :

```
$ git rebase -i master
```

Créer une branche lors d'une restauration

On peut créer une branche lors d'une restauration d'un fichier en se plaçant préalablement dans la branche et en utilisant l'option -b : Command git

```
$ git checkout bfbfefa4d9116beff394ad29e2721b1ab6118df0 -b version_2
```

```
  Switched to a new branch "version_2"
```

Les Tags

Les tags sont très pratiques pour marquer une version bien spécifique. Par exemple, pour releaser une version 1.0, il est possible de créer un tag et de s'en resservir par la suite pour télécharger cette version particulière. On peut tagger ce que l'on veut, ça permet de se repérer facilement.

Créer un Tag

```
$ git tag '1.0'
```

Notes :Il est possible de signer les tag via GPG

Lister les tags

Pour lister les tags disponibles :

```
$ git tag
```

Supprimer un tag

Pour supprimer un tag:

```
$ git tag -d v0.1
$ git push origin :refs/tags/v0.1
```

Envoyer les tags sur le serveur

Une fois les tags créés en local, il est possible de les pousser sur le serveur:

```
$ git push --tags
```

Gérer les conflits

Il est possible de gérer les conflits de façon interactive :

```
$ git mergetool
```

Résolution automatique de conflits

Il est possible de demander à git de résoudre automatiquement des problèmes sur lesquels il serait déjà tombé :

```
$ git config --global rerere.enabled 1
```

Utiliser git pour trouver un bug

Git-bisect permet de découvrir exactement à quelle moment elle est regression de code, quel est le commit correspondant, pour avoir une idée très précise de sa cause.

Méthode empririque

Indiquer un commit présentant le dysfonctionnement (HEAD), et un commit passé, n'importe lequel, qui ne la présente pas.

Git se place alors dans l'historique de développement, pile au milieu entre les deux commit. Après vérification, ndiquer à git si la régression apparaît ou pas. Et on recommence. Chaque fois, on divise par 2 l'intervalle des commits ayant potentiellement introduit le bug, jusqu'à débusquer le fautif.

Exemple:

```
$ git bisect start
$ git bisect bad <mauvaise_version>
$ git bisect good <bonne_version>
```

```
  Bisecting: 8 revisions left to test after this
```

8 commits ont potentiellement introduit la régression. On indique si le commit actuel est correct ou pas :

```
$ git bisect good # ou git bisect bad
```

```
  Bisecting : 4 revisions left to test after this
```

Et on continue, jusqu'à ce qu'on se trouve le mauvais commit à l'origine de la regression :

```
37b4745bf75e44638b9fe796c6dc97c1fa349e8e is first bad commit
```

A tout moment, on peut obtenir un historique de votre parcours :

```
$ git bisect log
```

Si a un moment précis, on ne souhaite ne pas tester un commit en particulier, pour n'importe quelle raison, utiliser la commande :

```
$ git bisect skip
```

Utilisation d'un script

Il est possible d'automatiser la recherche en utilisant des scripts:

```
$ git bisect start HEAD <bad_commit> --
$ git bisect run script
```

Note : le script doit renvoyer 0 si le code est correct, 1 s'il est incorrect, et 125 s'il est intestable (git skip). Pendant la bisection, git créé une branche spéciale dédiée. Repasser sur la branche de développement une fois la régression repérée. Il est possible de le faire simplement en tapant :

```
$git bisect reset
```

Connaître ligne par ligne les commits

Ceci est la solution de taper sur les doigts d'une personne qui a réaliser un commit. Entre autre, vous avez la possibilité de connaître ligne par ligne qui a écrit quoi : Command git

```
$ git blame <fichier>
```

```
  ^a35889c (Deimos 2010-04-20 17:36:29 +0200   1) <?php
  ^a35889c (Deimos 2010-04-20 17:36:29 +0200   2) class DeleteHistory
extends SpecialPage
  ^a35889c (Deimos 2010-04-20 17:36:29 +0200   3) {
  724f724d (Deimos 2011-06-14 23:15:40 +0200   4)   function __construct()
  724f724d (Deimos 2011-06-14 23:15:40 +0200   5)   {
  724f724d (Deimos 2011-06-14 23:15:40 +0200   6)       // Need to belong to
Administor group
  724f724d (Deimos 2011-06-14 23:15:40 +0200   7)       parent::__construct(
'DeleteHistory', 'editinterface' );
  724f724d (Deimos 2011-06-14 23:15:40 +0200   8)
```

```
wfLoadExtensionMessages('DeleteHistory');
```

Déplacer un dossier et son historique vers un autre repository

On peut parfois avoir besoin de recréer un autre repository vierge pour contenir le dossier d'un autre repository. Ca a été mon cas pour l'extension DeleteHistory de Mediawiki qui j'ai créer. Au début, j'avais un repository appelé 'mediawiki_extensions' ou j'avais 2 extensions dedans. Et puis avec le temps et les version de Mediawiki qui avançaient, il était préférable de faire une séparation par plugins tout en gardant l'historique. Je vais donc vous expliquer comment j'ai fais (j'ai suivit cette documentation[2]).

Nous allons travailler en local avec mon ancien repository que nous allons appeler oldrepo et mon nouveau repository newrepo.

Attention: Faites un repository temporaire car il sera supprimé à la fin

Cloner le repo :

```
$ git clone git://git.deimos.fr/oldrepo.git
$ git clone git://git.deimos.fr/newrepo.git
```

Filtrer sur le dossier qui nous intéresse, appelons le folder2keep :

```
$ git filter-branch --subdirectory-filter folder2keep -- -- all
```

Tous les fichiers et dossiers de folder2keep sont à la racine durepository. Si vous le souhaitez, vous pouvez créer un dossier et tout placer dedans, mais ceci n'est pas obligatoire :

```
$ mkdir new_directory/
$ git mv * new_directory/
```

Remplacer * par tout élément que l'on souhaite placer dedans

Commiter :

```
$ git commit -m "Collected the data I need to move"
```

Faire une référence locale

Se placer dans le dossier et faire une référence locale entre les 2 :

```
$ cd ../newrepo/
```

```
$ git remote add oldrepo ../oldrepo/
```

Rappatrier les sources

Rappatrier les sources, créer la branche principale et merger le tout

```
$ git fetch oldrepo
$ git branch oldrepo remotes/oldrepo/master
$ git merge oldrepo
```

Transférer l'historique

Transférer l'historique et faire un peu de ménage :

```
$ git remote rm oldrepo
$ git branch -d oldrepo
$ git push origin master
```

Le repository temporaire (vieux) peut être maintenant supprimé.

Hooks

Il existe des hooks dans Git permettant de faire des pre ou post traitement. Voici un use case permettant d'effectuer une action (lancer une commande via ssh) lorsqu'un tag arrive sur le serveur. L'idée est de pouvoir en fonction d'un tag (exemple: prod ou preprod) déployer une nouvelle version d'un soft sur X serveurs. Voici la liste des options qui vont être nécessaire:

- Créer un hook dans le repository
- Créer un utilisateur dédié si vous utilisez Gitlab/GitHub/Gitolite
- Générer une clef privée SSH avec l'utilisateur 'git'
- Copier via ssh-copy-id vers les serveurs distant la clef de l'utilisateur 'git' (généralement www-data)
- Créer un script de déploiement et mettre les bons droits
- Cloner le repository sur tous les serveurs nécessaire

Configuration File repo.git/hooks/post-receive

```
#!/bin/bash
# Create your environment with DNS/IP servers separated by spaces
name=( list array )
prod=( X.X.X.X Y.Y.Y.Y )
preprod=( Z.Z.Z.Z )
# Folder to deploy on client side
folder='/var/www/cloned_repository'
# Set SSH remote username
ssh_username='www-data'
```

```
# Log file
logfile='/var/log/git-deploy-hook.log'
################################################################################
# get infos
read oldrev newrev refname
# vars
tag=`echo $refname | cut -d/ -f3`
environment=`echo $tag | cut -d- -f1`
# function to call distant git deployments
deploy_new_version() {
    declare -a servers=("${!1}")
    echo $(date +'%Y/%m/%d - %H-%M-%S ') "Begin deployment hook for
$environment env" >> $logfile
    for server in $(seq 0 $((${#servers[@]} - 1))) ; do
        echo "Deploying $tag" >> $logfile
        ssh $ssh_username@${servers[$server]} /usr/bin/git_deploy.sh $tag
$folder 2>&1 >> $logfile
    done
    echo $(date +'%Y/%m/%d - %H-%M-%S ') "Deployment hook finished for
$environment env" >> $logfile
}
# check if a tag has been pushed
if [[ -n $(echo $refname | grep -Eo "^refs/tags/$environment") ]]; then
    echo "Git hook is called for $environment environment"
    if [ ${!environment[0]} ] ; then
        # Deploy the correct environment
        deploy_new_version $environment[@]
    else
        echo "Error, $environment is not recognized as an existing
environment"
        exit 1
    fi
fi
```

Modifier les environnements avec ceux voulu pour que quand un tag arrive avec le nom du tag en question, il y ai une action qui soit déclenchée derrière. Par exemple pour déployer sur les serveurs de prod, il faudra mettre un tag de type “prod-v1.0”.

Switcher avec l'utilisateur git (ou celui qui a les droits) et copiez la clé publique vers tous les serveurs sur lesquelles vont devoir agir les tags:

```
$ su - git
$ ssh-copy-id www-data@x.x.x.x
```

Créer le script de déploiement:

Configuration File /usr/bin/git_deploy.sh

```
#!/bin/bash
logfile='/var/log/git-deploy-hook.log'
echo $(date +'%Y/%m/%d - %H-%M: ') 'Begin deployment hook' >>$logfile
```

```
cd $2
git fetch origin -v >> $logfile 2>&1
git fetch origin -v --tags >> $logfile 2>&1
git stash drop >> $logfile 2>&1
git checkout -f tags/$1 >> $logfile 2>&1
echo $(date +'%Y/%m/%d - %H-%M: ') 'Deployment finished' >> $logfile
```

Pour rajouter toutes les commandes Il faut copier ce script sur tous les serveurs cible et lui donner les droits d'exécution.

Créer le fichier de logs:

```
$ touch /var/log/git-deploy-hook.log
$ chow www-data. /var/log/git-deploy-hook.log
$ chmod 755 /usr/bin/git_deploy.sh
```

Cloner les repository avec les utilisateurs finaux:

```
$ cd /var/www
$ git clone git@gitlab/cloned_repository.git
$ chown -Rf www-data. /var/www/cloned_repository
```

Pusher un tag sur un commit:

```
$ git tag -a prod-v1.0 -m 'production version 1.0' 9fceb02
$ git push --tags
```