Tutoriel git
UE 2I013

Gabriella Contardo – Nicolas Baskiotis
Sorbonne Université

2019

Introduction

git est un logiciel de gestion de versions décentralisé. Ses principales caractéristiques sont les suivantes :

- gestion de versions : il permet de stocker des fichiers tout en gardant l'historique de toutes les modifications. Un *commit* correspond à un point d'étape, une sauvegarde de l'état courant des fichiers. À tout moment dans le futur, on peut comparer l'état actuel d'un fichier à son historique, par rapport à n'importe quel moment où il a été *commité*. Il est également possible de comparer différents *commits* entre eux, de restaurer des anciennes versions/commits. Les sauvegardes sont incrémentales, i.e. seules les différences entre les commits sont enregistrés à chaque fois ce qui permet de ne pas dupliquer des informations inutiles et donc de ne pas alourdir le stockage avec des informations redondantes.
  Attention !! ce n'est pas le cas pour les fichiers binaires ! Comme un fichier binaire change souvent radicalement entre plusieurs versions, à chaque commit le fichier est sauvegardé entièrement. Il faut donc limiter autant que possible de gérer des fichiers binaires (on ne stocke quasiment jamais les binaires compilés sauf quand il s'agit de release).

- collaboratif : plusieurs personnes peuvent modifier et travailler sur les fichiers, **git** utilise une fusion intelligente des modifications pour mettre à jour la nouvelle version lors des commits. Il se peut qu'il y ait des conflits, si par exemple des modifications sont faîtes à un même endroit dans un fichier; dans ce cas, il faut modifier à la main les fichiers pour résoudre les conflits.

- décentralisé : contrairement à **subversion**, **git** n'a pas de serveur centralisé qui jouerait le rôle de référent. Un *dépôt* peut être *cloné* (sur un même ordinateur ou de manière distante), les modifications peuvent se faire dans les différents dépôts, puis être fusionné si besoin (par un *pull request*, une demande de fusion d'une version vers un autre dépôt qui joue le rôle de dépôt principal/officiel).

Il est important de noter que **git** peut être utilisé sans dépôt distant, uniquement sur son propre ordinateur comme gestion de versions de son propre code/projet. Il peut bien sûr être utilisé de manière décentralisé.
Il existe principalement deux services en ligne proposant **git** (ainsi que différents outils DevOps), `http://github.com` (payant pour une partie des fonctionnalités) et `http://gitlab.com` (open-source et pouvant être installé sur son propre serveur). Nous utiliserons dans la suite **github**, plus léger et plus répandu.
En complément de ce tutoriel, infos utiles, vocabulaire et commandes, en français: `http://christopheducamp.com/2013/12/15/github-pour-nuls-partie-1/`
Et la documentation officielle:

`https://git-scm.com/book/fr/v1/Les-bases-de-Git-Dmarrer-un-dpt-Git`

Attention : il faut configurer le proxy sur les machines de la PPTI pour pouvoir accéder à github.com. Le protocole est en https qui n'est pas configuré par défaut. Avant la première utilisation de **git**, il vous faudra définir de manière pérenne le proxy en ajoutant à la fin du fichier `~/.bashrc` la ligne

export https_proxy=proxy:3128

Ensuite, démarrez un **nouveau** terminal.

1 Commandes de base (en local)

Les commandes de cette section n'agisse que en local (sur l'ordinateur que vous utilisez). Créer un répertoire de test, puis dans la console/terminal, tester les commandes pour commencer à vous familiariser avec **git**.

- `>git help` : aide et récapitulatif des commandes de bases;
- `>git init` : transformer un répertoire (dossier) classique en dépôt (*repository*) git. C'est la première commande à exécuter afin d'initialiser un repo.

  Attention : les fichiers qui seront mis dans ce répertoire ne sont pas gérés par **git** par défaut ! Il faut signaler à **git** quels fichiers feront partie du projet (par la commande ci-dessous).
- `>git add <nom_fichier>` : "Linker" un fichier à votre repo git.
  Cette commande signale à git que le fichier doit être "suivi" (indexé) et également que le fichier est prêt à être ajouté (lors de la création ou d'une modification).

  Attention: Cette commande indexe la version du fichier au moment où elle est appelée; si vous re-modifiez "mon_fichier" ultérieurement sans appeler "add" de nouveau, les modifications ne seront pas prises en compte (voir exemple plus bas), seule la version qui date du dernier appel de `add` est considérée.
- `>git add .` : pour indexer tous les éléments du repo courant.
- `>git commit -m "<Message>"` : soumettre les modifications de fichiers - i.e enregistrer un instantané des versions des fichiers suivis.

  Attention : c'est la seule commande qui permet effectivement d'enregistrer un historique de l'état actuel. La commande `add` ne fait qu'indiquer à **git** d'indexer les fichiers, mais ne réalise pas de capture de l'état courant. Il faut impérativement faire un `commit` après une commande `add` pour demander à **git** d'enregistrer l'état courant. Dans ce cas là, le commit n'envoie que les modifications enregistrées par les différents appel à `add`.
  Une option particulière : `>git commit -am "Message"` qui effectue un `add` sur tous les fichiers déjà indexés (pas les nouveaux fichiers par contre !) puis un `commit`.
- `>git status` : obtenir l'état du repo et des indexations en cours. Très utile pour savoir ce qui a été modifié, ce qui a été ajouté et les fichiers ignorés.
- `>git rm <nom_fichier>` : supprime un fichier de l'index et du répertoire (`git rm --cached nom_fichier` pour supprimer uniquement de l'index). Ne jamais supprimer un fichier indexé par un autre moyen, l'information ne serait pas transmise à **git**.
- `>git diff` : montre les différences entre l'état courant et le dernier commit. `git diff <A> <B>` : les différences entre les commit A et B.
- `>git log` : historique des commits.
- `>git checkout <nom_fichier>` : Réinitialiser un fichier à sa dernière version commitée.

  Attention : Ceci efface toutes les modifications réalisées depuis le dernier commit sur votre fichier !
- `>git reset HEAD` : annule les dernières commandes `add` depuis le commit indiqué (`HEAD` pour le dernier commit). Ne modifie pas les fichiers. Avec l'option `--hard`, permet de revenir entièrement à l'état du commit indiqué. Les fichiers sont modifiés, tout votre travail depuis le dernier commit est perdu.
- Le fichier `.gitignore` : contient les expressions régulières des fichiers qu'il ne faut pas indéxer, un par ligne (par exemple `*.pyc` pour ignorer les fichiers python compilés , `tmp/` pour ignorer un répertoire temporaire, ...).
- `>git clone <adresse_du_repository/nom_du_repo>` : copier un repo existant dans un nouveau répertoire (e.g une adresse github ou un repo local). Cela crée un dossier "nom_du_repo" qui contient la dernière version du repo.

Un petit exemple pour commencer à jouer, tester intensivement toutes ces commandes !

```
>mkdir mon_repo_git
>cd mon_repo_git
### Configurer git avec son Prenom Nom et adresse email
>git config --global user.name "Prenom Nom"
```

```
>git config --global user.email "email@email.org"
### Initialiser un repertoire
>git init
Initialized empty Git repository in mon_repo_git/.git/
### Git vide initialisé dans mon_repo_git/.git/
### Creation d'un fichier et ajout au depot
>echo "Mon premier depot" > README.txt
>git status
### Les fichiers en rouge sont les fichiers non suivis.
### Le fichier qui vient d'être créé n'a pas encore été indexé.
>git add README.txt
>git status
### Le fichier est en vert : les modifications ont été indexées.
### Par contre, toujours aucun fichier dans le repo.
>git commit -m "Premier commit"
>git status
### Premier vrai commit. Le status indique que tout est à jour.
>echo "Modif" >> README.txt
>git status
### De nouveau une ligne rouge qui indique des changements par rapport au dernier commit.
>git add README.txt
>git status
>echo "2nd Modif" >> README.txt
>git status
### Le meme fichier et en vert, et en rouge : en vert, des modifications ont été indexées;
###en rouge, le fichier a été modifié après la dernière indexation (par add).
>git commit -m "2eme commit"
>git status
>git diff
### Montre la différence entre l'état courant et le fichier indexé.
>git log
### Montre les commits réalisés.
```

2 Dépôts distants

Les commandes précédentes agissaient uniquement en local (création et utilisation d'un dépôt local). L'intérêt des logiciels de gestion de version est de pouvoir être décentralisés et accessibles par réseau. Pour plus d'infos sur le fonctionnement et l'intérêt des dépôts distants : `http://tinyurl.com/gopz8bf`.
Vous aurez besoin de créer un compte sur `http://github.com` pour la suite et de configurer le proxy (sur les ordinateurs de la PPTI).

- `>git remote -v` : visualiser les dépôts distants auxquels vous êtes liés.
- `>git remote add <depot_distant> <adresse_du_depot>` : ajouter un dépôt distant. Il est d'usage d'avoir un dépôt nommé `origin`, le dépôt principal.
- `>git pull <depot_distant> <nom_branche>` : récupérer et fusionner les commits de la branche `<nom_branche>` (la branche principale d'appelle `master`) à partir du dépôt distant vers le dépôt local.
- `> git fetch <depot_distant>` : récupérer sans fusionner les données du dépôt distant. Cela récupère aussi les branches du dépôt.
- `>git push <depot_distant> <nom_branche>` : pousser votre travail réalisé sur la branche `<nom_branche>` vers le dépot distant (par défaut *origin*). Il faut que vous ayez les droits en écriture pour `push`.
- `>git stash` : quand plus rien ne marche parcequ'il y a trop de conflits entre le local et le distant, cette commande magique permet de nettoyer votre dépôt local sans perdre le travail effectué. Des commandes comme `git stash list` ou `git stash show` permettent d'inspecter les différences.

Créer un compte `github`, puis tester les commandes.

```
### configurer le proxy
> export https_proxy = proxy:3128
```

```
### Configuration du push (match toutes les branches)
>git config --global push.default matching
### Cloner un depot
> git clone https://github.com/baskiotisn/test.git
> git remote -v
### origin est fixé par défaut au repo cloné
### Modification des fichiers
> echo "print(1)" >> test.py
> git commit -am "modif"
### Le dépôt local a bien été modifié.
> git push origin master
### Erreur ! vous n'avez pas les droits d'écriture...
### Creer un repo git et ajouter le en raccourci :
> git remote add monrepo https://github.com/username/repo.git
### L'option -u permet de lier la branche actuelle à celle du remote,
### à ne faire qu'une fois.
> git push -u monrepo master
### La ca marche !
### Modifier des choses depuis github, puis :
> git pull monrepo
```

3 Branches

Vous pouvez créer avec `git` des *branches* dans votre code, ce qui permet de travailler sur différents aspects en parallèle en gardant un code commun stable (la branche *master*). Chaque branche évolue de façon indépendante (une par exemple pour corriger un bug, l'autre pour rajouter une fonctionnalité en cours de développement, ...). Une branche peut ensuite être fusionner (merge) à la branche principale *master*.

- `>git branch -v` : obtenir la liste des branches existantes, celle précédée d'une étoile étant celle où vous vous trouvez.
- `>git branch -a` : obtenir la liste des branches existantes en local et en remote.
- `>git branch <nom_branche>` : pour créer une nouvelle branche nommée `<nom_branche>`.
- `>git checkout <nom_branche>` : pour basculer sur la branche nommée `<nom_branche>`.
- `>git merge <autre_branche>` : Pour fusionner la branche `<autre_branche>` dans la branche courante. Si un fichier a été modifié sur les deux branches, il peut y avoir un conflit qu'il faut alors résoudre à la main. Exemple:

```
### Creer une nouvelle branche 'dev'
>git branch dev
### Se deplacer sur dev
>git checkout dev
### Effectuer des changements et un commit
>echo "Modif from dev" >> README.txt
>git commit README.txt -m 'First commit on branch dev'
### Se déplacer dans la branche master (principale)
### et effectuer un autre modification
>git checkout master
>echo "Modif from master" >> README.txt
>git commit README.txt -m 'Commit on branch master'
### Fusionner le travail fait sur dev dans master
>git merge dev
### Le même fichier a été modifié sur les 2 branches,
### il faut résoudre le conflit à la main
>nano README.txt
### On peut maintenant indiquer à git que le conflit est résolu
### et faire un commit pour la fusion des branches
>git add README.txt
>git commit
```

```
### Le travail effectué sur la branche dev est maintenant
### également sur la branche master.
```

Après le merge, vous pouvez effacer la branche : `>git branch -d branche_machin`