# GIT et GITHUB

## Notes

Jérôme JONDEAU

le 28 septembre 2020

# Sommaire

# Informations

Les informations typographiées avec une police de `type machine à écrire` représentent des noms ou des entrées de menu.

Dans un lecteur de fichier *pdf* les parties de texte présentées en caractères bleus sont des liens hypertextes pointant vers un partie du document.

# Chapitre 1

# Préparation et démarrage

## 1.1 Informations

Cette section présente l'application Git et indique comment préparer son utilisation.

### 1.1.1 Présentation

Git est un système de gestion de versions, de l'anglais **V**ersion **C**ontrol **S**ystem, communément raccourci en VCS. Selon leur principe de fonctionnement, les VCS se répartissent en trois systèmes principaux :

1. local ou VCS,
2. centralisé ou **C**entralized VCS,
3. distribué ou **D**istributed VCS.

Git est un VCS. Contrairement à d'autres solutions, il n'est pas prévu pour suivre un flux de modifications mais plutôt un flux d'instantanés, où chaque instantané peut être considéré comme « un mini système de fichiers ». Les opérations sont donc locales, chaque validation est signée en SHA-1 et nommée par sa signature.

Il n'existe que deux statuts de données : non suivi et suivi. Au cours de son suivi de version, une donnée aura l'un des trois états suivants :

modifié le contenu a été modifié, il n'a pas encore été ni indexé ni validé ;

indexé : le contenu modifié a été pris en compte et est en attente de validation ;

validé : le contenu indexé a été enregistré dans la base de données. Il servira de référence pour le prochain instantané.

En correspondance, ces trois états reflètent trois zones de Git :

le répertoire de travail : la zone de modification des données ;

la zone d'index : les données qui seront utilisées pour le prochain instantané ;

le répertoire .git : l'ensemble des méta-données et la base de données.

Le mode opératoire global de suivi est donc lui aussi logiquement réparti en trois étapes :

1. modification des données,
2. sélection et indexation des données modifiées,
3. validation des données indexées.

### 1.1.2 CLI et GUI

GIT est utilisable en ligne de commande et au travers d'interfaces graphiques. Bien qu'utiliser la ligne de commande puisse paraître d'un autre âge, elle offre deux avantages importants :

1. les commandes sont identiques sur toutes les plateformes,
2. toutes les commandes sont disponibles.

En comparaison, les interfaces graphiques proposent un aspect et un fonctionnement spécifiques à chaque application. Mais leur véritable inconvénient est de ne proposer qu'un sous-ensemble – souvent restreint – des commandes disponibles.

## 1.2 Configuration initiale

La version GIT est obtenue, sous GIT BASH et GIT CMD, par la commande `git --version`.

Cette section présente la configuration de GIT pour LINUX et indique la liste des fichiers de configuration pour MICROSOFT WINDOWS [1] .

### 1.2.1 Les fichiers de configuration sous Linux

Le fichier `/etc/gitconfig` définit la configuration globale de GIT, destinée à tous les utilisateurs. La commande `git config --system --list` affiche les paramètres de ce fichier.

Le fichier `~/.gitconfig` définit la configuration spécifique d'un utilisateur. La commande `git config --global --list` affiche les paramètres de ce fichier.

Enfin, un fichier `.git/config` est présent dans chaque dossier de dépôt pour en définir la configuration et la structure.

### 1.2.2 Les fichiers de configuration sous Microsoft Windows

Le fichier `%windir%:\ProgramData\Git\config` définit la configuration globale de GIT, destinée à tous les utilisateurs.

Le fichier `%homepath%:\.gitconfig` définit la configuration spécifique d'un utilisateur.

Enfin, le fichier `.git\config` est présent dans chaque dossier de dépôt pour en définir la configuration et la structure.

### 1.2.3 Configuration initiale de l'utilisateur

Il est nécessaire de définir l'identité de l'utilisateur en deux commandes :

1. `git --config user.name "John Doe"`
2. `git --config user.email "john.doe@example.com"`

Par défaut sous LINUX, GIT utilise l'éditeur de fichier VIM. La commande suivante permet de définir l'éditeur EMACS : `git --config --global core.editor emacs`.

1. A partir de MICROSOFT WINDOWS 7

### 1.2.4 Affichage des paramètres

L'affichage de l'ensemble des paramètres – global et utilisateur – est obtenu avec la commande `git config --list`. L'éventuel double affichage de certains paramètres est dû au fait qu'ils sont successivement lus dans les deux fichiers de configuration : dans ce cas la dernière valeur affichée prévaut.

L'affichage d'un paramètre est obtenu par la commande `git config <parametre>`. Par exemple, le nom de l'utilisateur est affiché avec la commande `git config user.name`.

## 1.3 Obtenir de l'aide

Sous Linux l'aide pour une commande est accessible de trois façons :

1. `git help <commande>`
2. `git <commande> --help`
3. `man git-<commande>`

# Chapitre 2

# Démarre ou cloner un dépôt

## 2.1 Démarrer un dépôt Git

### 2.1.1 Depuis un dossier existant

Le dossier courant est celui dont on se propose de suivre les données (il est ici supposé que ce dossier esdt vide). La commande initiale `git init` créé le sous-dossier `.git` qui contiendra les méta-données et la base de données du dépôt. Cette commande ne doit être exécutée qu'une seule fois, au moment où Git prend en compte le dossier.

La commande `git status` récapitule l'état de suivi des données. Ici l'état indique "dossier propre" car le dossier est vide.

Créons le fichier `README` : la commande `git status` indique que fichier `README` n'est pas suivi. La commande `git add README` indexe ce fichier : la commande `git status` indique maintenant qu'il est suivi.

La commande `git commit -m "commit initial"` valide le fichier indexé pour créer un instantané, référencé par une signature numérique SHA-1. La commande `git status` indique de nouveau "dossier propre".

### 2.1.2 Cloner un dépôt

Le clonage va rapatrier le dépôt distant dans le répertoire courant en y créant un dossier ad-hoc.

Admettons que le dépôt à cloner soit `https://github.com/libgit2/libgit2`. La commande `git clone https://github.com/libgit2/libgit2` procède aux opérartions suivantes :

1. création locale du dossier `libgit2`,
2. récupération de la base de données du dépôt,
3. initialisation du sous-dossier .git,
4. extraction de la dernière version.

Pour personnaliser le dossier local en `monlibgit2`, la commande devra être modifiée pour spécifier le dossier, soit `git clone https://github.com/libgit2/libgit2 monlibgit2`.

Dès lors, le suivi de version est identique à celui présenté à la section 2.1.1.

# Chapitre 3

# Les commandes de base

## 3.1 Le statut court

La commande `git status -s` affiche un état synthétique du suivi de données, sur deux colonnes : la colonne de gauche représente l'état de l'index, celle de droite celui du répertoire de travail. Ci-dessous les différents états de suivi :

"_M" : fichier indexé modifié mais pas encore indexé

"MM" : fichier modifié et indexé puis de nouveau modifié

"M_" : fichier modifié et indexé

"AM" : nouveau fichier indexé puis modifié

"A_" : nouveau fichier indexé

"??" : fichier non suivi

## 3.2 Indexer des fichiers

Les commandes suivantes présentes des exemples courants d'indexation :

`git add readme` : indexer le fichier `readme`

`git add *.c` : indexer tous les fichier d'extension `c`

`git add nom_dossier` : dans le cas d'un dossier, tous les éléments du dossier sont indexés, en mode récursif

## 3.3 Ignorer des fichiers

## 3.4 Fichiers d'exclusion

Un fichier d'exclusion, à nommer `.gitmore`, liste les modèles de nom de fichiers ou dossiers à exclure du suivi. Un fichier d'exclusion s'applique récursivement sur toute la structure du dossier de travail ou jusqu'à rencontrer un nouveau fichier d'exclusion.

### 3.4.1 Définir les exclusions

Chaque ligne du fichier `.gitmore` définit une exclusion, comme indiqué ci-dessous.
— une ligne vide est ignorée
— une ligne de commentaire, qui doit débuter par #, est ignorée
— ensemble de caractères :
  — pour les noms de fichiers ou de dossiers

    - `[bB]`uild exclut `build` et specBuild
- patron de fichier :
    - `dir1/dir2/file.ext` :
        - exclut le fichier file.ext en récursif
        - à partir du dossier dossier de travail/dir1/dir2/
    - `otherfile.ext` :
        - exclut le fichier otherfile en récursif
        - à partir du dossier dossier de travail
    - `doc/*.apk` :
        - exclut tous fichiers d'extension `apk`
        - à partir du dossier `doc` en mode résursif
    - `/*.apk` :
        - exclut tous fichiers d'extension `apk`
        - uniquement dans le dossier courant (sans récursion)
    - exception à l'exclusion :
        - `!*`
        - `!a.apk`
        - exclut tous les fichiers d'extension `apk` sauf `a.apk`
        - uniquement dans le dossier courant (sans récursion)
- patron de dossier :
    - `dir1/` : exclut le dossier dir1 lui-même et en récursif
    - il faut spécifier un dossier par lig
- : `**/dir1/` :
    - exclut le dossier `dir1` en mode récursif
    - sans exclure les fichiers `dir1`
- : `dir1/**/dir2` :
    - exclut le dossier `dir2`
    - quelqu soit le nombre de dossiers entre `dir1` et `dir2`
- patron commun fichier et dossier :
    - `elem` : exclut tous les fichiers et tous dossiers en mode résursif

### 3.4.2 Exemples

```
# pas de fichier .a
.a

# conserver lib.a malgré la règle précédente
!lib.a

# ignorer le fichier TODO à la racine du projet
/TODO

# ignorer tous les fichiers dans le dossier build
build/

#ignorer doc/note.txt mais pas doc/server/arch/txt
doc/*.txt
```

## 3.5 Inspecter les modifications non indexées et indexées

### 3.5.1 Les modifications non indexées

La commande `git diff` compare le contenu du dossier de travail avec la zone d'index pour établir la liste toutes les modifications qui n'ont pas encore été indexées.

Si toutes les modifications ont été indexées, la liste est vide. Sinon elle affiche les fichiers modifiés et les modifications apportées à chacun d'entre eux. Il est possible de visualiser cette liste sous forme graphique avec la commande `git difftool`.

La commande `git difftool --tool-help` liste à son tour les applications disponibles pour cet affichage.

### 3.5.2 Les modifications indexées

La commande `git diff --staged` compare la zone d'index avec le dernier instantané pour établir la liste de toutes les modifications indexées. Ces modifications seront celles prises en compte lors de la prochaine validation.

## 3.6 Valider les modifications

Deux méthodes permettent de valider les données indexées.

### 3.6.1 Via la commande d'édition

La commande `git commit [-v]` lance l'éditeur défini dans le configuration Git pour permettre la saisie de l'intitulé de commit.

Sans le paramètre `-v` la liste des nom de fichiers à valider est afficher. Avec le paramètre un `git diff` de chaque fichier indexé est affiché.

Une fois l'intitulé du commit saisi sur la première ligne, le fichier édité doit être fermé avec sauvegarde. Le `commit` est alors automatiquement réalisé avec l'intitulé saisi.

### 3.6.2 Via la commande directe

La commande `commit -m "intitulé"` réalise directement le commit en l'intitulant avec le texte indiqué en dernier paramètre.

La commande `commit -am "intitulé"` indexe automatiquement tous les fichiers déjà en suivi de version et produit directement le commit avec le texte indiqué en dernier paramètre.

## 3.7 Supprimer ou renommer un fichier

### 3.7.1 Supprimer un fichier

Deux cas de figure sont envisageables : demander à Git de supprimer le fichier ou le supprimer directement de dossier de travail.

#### Suppression par Git

GIT supprime le fichier du dossier de travail, les éventuelles informations de suivi puis indexe automatiquement la suppression.

Si ce fichier n'est pas indexé, la commande `git rm nomFichier` procèdera à la suppression. S'il est indexé, il faudra forcer la suppression avec la commande `git rm -f nomFichier`.

#### Suppression directe du fichier

La suppression directe d'un fichier dans le dossier de travail sera indiquée par la commande `git status`. Il faudra alors appeler `git rm [-f] nomFichier` pour indexer la suppression.

#### Un cas pratique

Comment supprimer le suivi d'un fichier, par exemple suite à une modification `.gitignore` ? En utilisant la commande `git rm --cached nomFichier`.

### 3.7.2 Renommer un fichier

La commande `git mv nom_origine nom_cible` renomme le fichier `nom_origine` en `nom_cible`. Cette commande est directement indexée. Les trois commandes suivantes seraient nécessaires pour produire le même résultat :

1. `mv nom_origine nom_cible`
2. `git rm nom_origine`
3. `git add  nom_cible`

## 3.8 Visualiser l'historique des validations

La commande `git log` affiche tous les `commit` réalisés, le dernier apparaîsant en premier. Cet affichage est réalisé par un outil de pagination.

La commande `git log` peut être assortie de très nombreux paramètres. Les plus utilisés sont listés ci-dessous :

-p : afficher les différences introduites pour chaque validation,

-x : limiter l'affichage aux `x` commits les plus récents,

–stat : afficher les statistiques résumées de chaque commit,

-Snom : limiter l'affichage aux commits qui ajoute ou retire du texte comportant `nom`,

–pretty : modifier le format des informations de sortie.

## 3.9 Modification d'un commit

Le `commit` erroné ne peut pas être modifié : il faut donc le remplacer.

### 3.9.1 Modifier l'intitulé d'un commit

La commande `git commit --amend` permet de saisir un nouvel intitulé, comme présenté à la section 3.6.1.

### 3.9.2 Ajouter un fichier au commit

Pour ajouter un `fichier` oublié, il suffit de l'indexer avec la commande `git add nomFichier` puis de lancer la commande `git commit --amend` : la saisie de l'intitulé est alors réalisé comme présenté à la section 3.6.1.

### 3.9.3 Désindexer un fichier

Il peut être nécessaire de désindexer un fichier dans l'objectif de réaliser un `commit` dédié à la modification qu'il apporte. La commande `git reset HEAD nomFichier` désindexe le fichier `nomFichier`. Il faut alors réindexer le fichier et créer le `commit` dédié.

### 3.9.4 Réinitialisation

#### Réparer une erreur non validée

La commande `git reset --hard HEAD` restaure le dossier de travail et l'index à l'état du dernier `git commit`. Les commandes `git diff` et `git diff --cached` ne rapporteront donc plus aucune différence.

Pour restaurer un seul fichier, deux commandes sont nécessaires :

1. `git checkout -- nomFichier` : restauration du fichier à la version de l'index (la commande `git diff` ne rapportera plus de différence),
2. `git checkout HEAD nomFichier` : restauration de l'index à la valeur du dernier `commit` (la commande `git diff --cached` ne rapportera plus de différence).

#### Réparer une erreur validée

Il faut créer un nouveau `commit` qui annule les changements du `commit` erroné. Deux exemples :
— la commande `git revert HEAD` annule les changements du dernier `commit (HEAD)` et lance l'édition d'un intitulé,
— la commande `git revert HEAD^` annule les changements de l'avant dernier `commit (HEAD^)` et lance l'édition d'un intitulé.

## 3.10 Connexion avec des dépôts distants

### 3.10.1 Afficher les connexions distantes

La commande `git remote` affiche la liste des dépôts distants enregistrés dans le dépôt local. Dans cette liste, le terme `origin` est le nom par défaut que Git attribue à un dépôt cloné.

La commande `git remote -v` ajoute l'url des dépôts à la liste affichée.

### 3.10.2 Ajouter une connexion distante sur un nom court

En enregistrant un dépôt distant il est possible de lui associer un non court qui pourra être utiliser dans toutes les opérations. Dans la liste `git remote`, le dépôt ne sera plus repéré par `origin` mais par son nom court.

La commande `git remote add pb https://github.com/pailboone/ticgit` associe le nom court `pb` au dépôt distant `\https://github.com/pailboone/ticgit`.

### 3.10.3 Inspecter les informations d'une connexion

La commande `git remote show nonDistant` affiche un récapitulatif complet des informations de la connexion.

### 3.10.4 Renommer un dépôt distant

La commande `git remote rename nonDistant nouveauNNomDistant` modifie le nom court d'une connexion. Les noms des branches sont automatiquement mis en jour.

### 3.10.5 Retirer un dépôt local

La commande `git remote rm nonDistant` retire la connexion distante `nonDistant`.

### 3.10.6 Récupérer et tirer un dépôt distant

La commande `git fetch nonDistant nomBranche` récupère les données mises à jour sur la branche `nomBranche` du dépôt distant et absentes du dépôt local. Ces données sont copiées dans une branche locale qui n'est pas une branche de travail. La commande `git merge` devra être utilisée pour fusionner ces informations sur une branche locale.

La commande `git pull origin master` opère la même opération mais procède également à la fusion des données récupérées dans la branche `master`. Elle équivant aux deux commandes successives `git fetch` et `git merge`.

### 3.10.7 Pousser vers un dépôt distant

La commande `git push origin master` envoie les modifications de la branche locale `master` du dépôt distant.

Pour que cette commande fonctionne il faut évidemment disposer de droits d'écriture sur le distant. De plus, toutes les modifications du distant doivent avoir été récupérées en local : dans le cas contraire il sera d'abord nécessaire de fusionner ces données en local – commande `git pull` – puis de pousser vers le distant.

## 3.11 Étiquetage

### 3.11.1 Introduction

Etiqueter un commit permet de localiser un état précis dans l'historique. La commande `git tag` affiche la liste alphabétiques des étiquettes définies. Il est possible de rechercher un motif particulier, par exemple `git tag -l 'v1.8.5*'`.

### 3.11.2 Créer des étiquettes

Il existe deux type d'étiquettes : annotée et légère.

### Étiquette annotée

Une étiquette annotée dispose d'une somme de contrôle, contient le nom et l'adresse de courriel du créateur, la date et un message d'étiquetage. Elle peut en outre être signée et vérifiée avec GNUPG : elle est stockée dans la base de données `git`.

La commande `git tag -a v1.4` lance l'éditeur défini dans la configuration GIT pour saisir le message d'étiquetage. La fermeture sauvegardée du fichier édité créé l'étiquette. La commande `git tag -a v1.4 -m 'version 1.4'` procède directement à la création de l'étiquette. Dans tous les cas le dernier `commit` est étiqueté par `v1.4`.

La commande `git show v1.4` affiche conjointement les informations de l'étiquette et du `commit` associé.

### Étiquettes légère

Une étiquette légère stocke uniquement la somme de contrôle d'un `commit` dans un fichier annexe. La commande `git tag v1.4g` créé l'étiquette légère `v1.4g` sur le dernier `commit`.

La commande `git show v1.4g` affiche uniquement les informations du `commit`.

### Étiqueter un commit précis

Il est possible d'étiqueter un `commit` précis, en indiquant tout ou partie de sa somme de contrôle. La commande `git tag -a v1.2 9fceb02` créé une étiquette annotée pour le commit dont la somme de contrôle débute par `9fceb02`.

### Partager les étiquettes

Par défaut, la commande `git push` ne pousse pas les étiquettes. Pour pousser l'étiquette `v1.2` il faut utiliser la commande `git push origin v1.2`. Pour pousser toutes les nouvelles étiquettes en une seule opération il faut utiliser la commande `git push origin --tags`.

## 3.12 Les alias

L'alias `git config --global alias.last 'log -1 HEAD'` affiche le dernier `commit`. L'appel sera `git last`.

L'alias `git config --global alias.ci 'commit'` exécute le `commit`. L'appel sera `git ci`.

L'alias `git config --global alias.st 'status'` affiche le statut.. L'appel sera `git st`.

# Chapitre 4

# Cas pratiques : suivi

## 4.1 Initialisation et suivi Git

Cette section résume un cas d'école : comment initialiser le suivi Git d'un dossier puis comment prendre en compte le suivi de la création et de la modification d'un fichier dans ce dossier.

### 4.1.1 Initialisation

1. se déplacer dans le dossier
2. `git init` : intialiser le suivi
3. `git status` : pour démonstration, afficher l'état actuel du suivi

### 4.1.2 Création et suivi d'un nouveau fichier

1. créer un fichier `nomFichier` dans le dossier
2. `git status` : le nouveau fichier est détecté comme non "suivi"
3. `git add nomFichier` : indexer le fichier (cqfd : prise en compte de son suivi)
4. `git commit -m "Premier fichier"` : valider la version du nouveau fichier
5. git log : afficher les informations du commit
6. `git status` : il n'y a plus de fichier à suivre

### 4.1.3 Modifier er suivre le fichier

1. modifier le fichier `nonFichier`
2. `git status` : la modification du fichier est détectée comme non "suivie"
3. `git add nomFichier` : indexer le fichier (cqfd : prise en compte de sa modification)
4. `git commit -m "Modification premier fichier"` : valider la version modifiée du fichier
5. `git log` : afficher les informations des commits
6. `git status` : il n'y a plus de modification à suivre

enditemize

### 4.1.4 Commandes push et pull

#### Commande pull

1. créer un dépôt GitHub

2. `git pull urlDepotGitHub` (url indiquée dans les paramètres GitHub du dépôt)
   a) Git contacte le dépôt distant, sur son url
   b) il créé un sous-dossier du nom du dépôt et s'y déplace
   c) il télécharge la base de données
   d) il extrait la base de données

### Commande push

1. créer un dépôt GitHub
2. se déplacer dans le dossier du dépôt local
3. `git remote add origin urlDepotGitHub` (url indiquée dans les paramètres GitHub du dépôt)
4. `git push origin master`