GIT

En solo, en 10 slides, pour les débutants

E. Cabrol

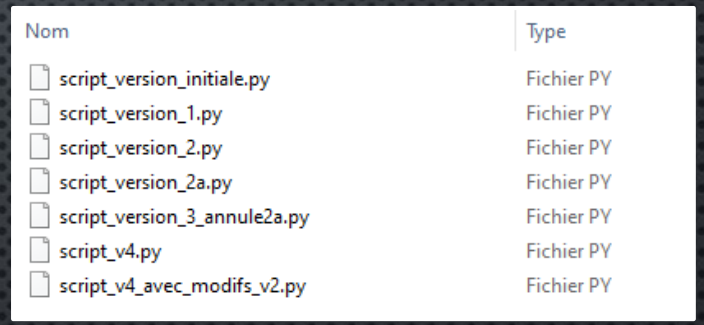
Juin 2020

INTRODUCTION

(NB: je ne suis pas un expert –même pas informaticien, c'est dire- et l'ouvrage de référence (Pro Git) pèse 530 pages. Donc méfiez-vous de ce que je dis ici !)

GIT est un outil de suivi de version, utile pour tous ceux qui écrivent plus ou moins régulièrement des lignes de code (*)

Son principal interet est d'éviter ce phénomène →



Grâce au suivi de version, on conserve l'historique des évolutions, on peut revenir quand on le souhaite à une étape donnée, et on peut créer des branches.

Je n'aborde dans ces pages que le travail « en solo », sans aborder la connexion à des dépôts distants et le travail collaboratif.

Les codeurs expérimentés se servant quotidiennement de git le trouvent sans doute intuitif, mais ce n'est pas forcément le cas du programmeur occasionnel dans mon genre, ou tout simplement du débutant. D'où ces quelques pages …

(*) Précision importante : ça fonctionne aussi sur des fichiers binaires, mais c'est surtout adapté pour des fichiers texte.

SE LANCER DANS LE GRAND BAIN

MÊME SI C'EST UN OUTIL DE GEEK, ÇA NE TOURNE PAS QUE SOUS LINUX, MAIS AUSSI SOUS WINDOWS, MAC, ETC … C'EST GRATUIT, OPEN-SOURCE, SOUS LICENCE GNU GPL.

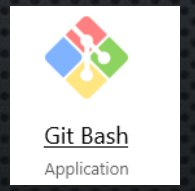
CA PEUT SE TÉLÉCHARGER ICI : HTTPS://GIT-SCM.COM/

LA VERSION WINDOWS A MÊME SON PROPRE SITE : HTTPS://GITFORWINDOWS.ORG/

LE PACKAGE EST LIVRÉ AVEC UNE IHM (*GIT-GUI*), MAIS IL EN EXISTE D'AUTRES.
DANS CES SLIDES L'OBJECTIF EST PLUTÔT DE COMPRENDRE LE PRINCIPE DE GIT, CE QUI IMPLIQUE JE PENSE DE DÉTAILLER QUELQUES-UNES DES COMMANDES EXISTANTES (ET DONC DE SE PASSER D'IHM …)

ON VA DONC SUPPOSER DANS LA SUITE QUE VOUS AVEZ RÉUSSI À :

- INSTALLER GIT
- LANCER GIT BASH (VERSION EN LIGNE DE COMMANDE)
- VOUS DÉPLACER DANS VOTRE DOSSIER DE TRAVAIL AVEC LA COMMANDE **cd** (CHANGE DIRECTORY)




```
ericc@LAPTOP-FQMPAS7S MINGW64 ~
$ cd test_git/
```

FAIRE SON PREMIER COMMIT

LA 1E COMMANDE A TAPER (UNE SEULE FOIS) EST :

```
git init
```

ELLE PERMET DE CRÉER LE DOSSIER .GIT DANS LEQUEL TOUTES LES EVOLUTIONS SONT TRACÉES. ICI UN SCRIPT EST DÉJÀ PRÉSENT, MAIS LE RÉPERTOIRE DE TRAVAIL (*WORKING TREE* DANS LA TERMINOLOGIE GIT) PEUT ÊTRE INITALEMENT VIDE.

ENSUITE

```
git add script.py
```

PERMET D'AJOUTER LE SCRIPT DANS LA *STAGING AREA*.

ENFIN

```
git commit –m "mon premier commit"
```

PERMET DE FINALISER CET « ARCHIVAGE » (*) SUR LE *REPOSITORY LOCAL*.

SI L'ON VEUT MODIFIER UN MESSAGE DE COMMIT JUSTE APRÈS L'AVOIR FAIT, OU AJOUTER UN FICHIER QU'ON AVAIT OUBLIÉ D'INDEXER, ON PEUT UTILISER L'OPTION `git commit --amend`

```
ericc@LAPTOP-FQMPAS7S MINGW64 ~
$ cd test_git/

ericc@LAPTOP-FQMPAS7S MINGW64 ~/test_git
$ ls
script.py

ericc@LAPTOP-FQMPAS7S MINGW64 ~/test_git
$ git init
Initialized empty Git repository in C:/Users/ericc/test_git/.git/

ericc@LAPTOP-FQMPAS7S MINGW64 ~/test_git (master)
$ git add script.py

ericc@LAPTOP-FQMPAS7S MINGW64 ~/test_git (master)
$ git commit -m "mon premier commit"
[master (root-commit) e408989] mon premier commit
 1 file changed, 1 insertion(+)
 create mode 100644 script.py
```

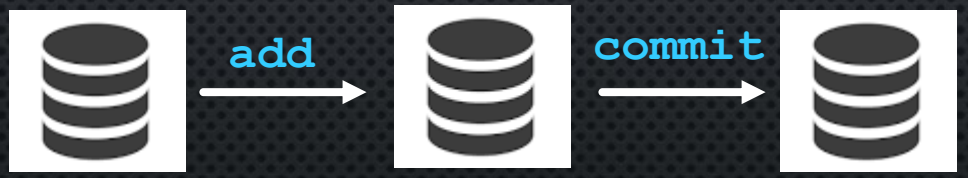
(*) la traduction n'est pas de moi mais du Grand Dictionnaire Terminologique Québecois

VOCABULAIRE ET CONCEPTS

OK, C'EST BIEN BEAU MAIS *STAGING AREA*, *REPOSITORY*, ÇA SIGNIFIE QUOI ?

LES TRADUCTIONS USUELLES SONT :

- **INDEX** POUR *STAGING AREA*. CA PEUT ÊTRE VU COMME UNE ZONE TEMPORAIRE. QUAND ON Y AJOUTE UN FICHIER, ÇA SIGNIFIE QU'ON SOUHAITE QU'IL SOIT ARCHIVÉ AU PROCHAIN COMMIT.
- **DÉPÔT** POUR *REPOSITORY*. LE DOSSIER .GIT CRÉÉ LORS DU GIT INIT EST NOTRE DÉPÔT LOCAL(*), C'EST LUI QUI CONTIENT LA TRACE DES MODIFICATIONS. ON N'IRA JAMAIS FOUILLER DEDANS « À LA MAIN », TOUTES LES INTERACTIONS PASSANT PAR LES COMMANDES DE GIT



(*) La connexion à des dépôts distants n'est pas traitée dans ce document, mais on peut tout à fait profiter de git uniquement « en local ».

STATUT D'UN FICHIER

Deux autres commandes sont fondamentales :

`git status` : permet de connaitre le statut des fichiers du dossier de travail

`git log` : permet de lister l'historique des opérations réalisées

Après mon premier commit du slide 4 on obtient cet affichage →

```
ericc@LAPTOP-FQMPAS7S MINGW64 ~/test_git (master)
$ git status
On branch master
nothing to commit, working tree clean

ericc@LAPTOP-FQMPAS7S MINGW64 ~/test_git (master)
$ git log
commit e408989122ab6e74738f1e17dad370f5b79c47f7 (HEAD -> master)
Author: Eric Cabrol <eric.cabrol@gmail.com>
Date:   Tue Jun 9 18:19:32 2020 +0200

    mon premier commit
```

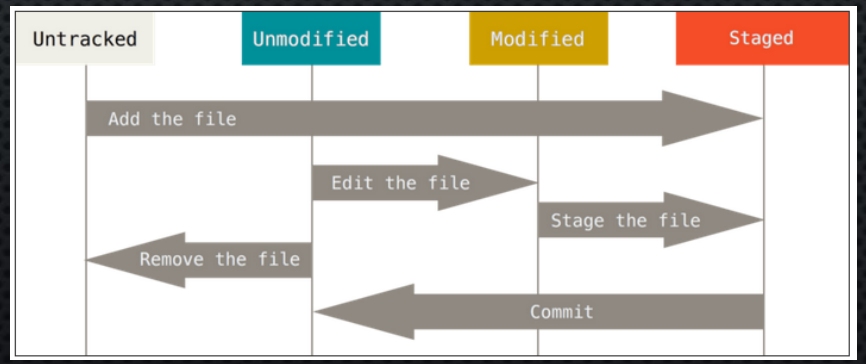
On voit après le mot commit une longue série de caractères (*) :

`commit e408989…`

C'est l'identifiant (unique) d'un commit. Pour pointer dessus on peut se contenter des premiers caractères

(**Important**) Un fichier peut être dans quatre états :

- untracked (non-suivi) : s'il n'a jamais été ajouté à l'index avec **`add`**
- modified (modifié) : s'il a été modifié depuis le dernier **`commit`**
- staged (indexé), juste après **`add`**
- Unmodified ou committed (archivé), juste après **`commit`**



(*) dont le nom technique est « somme de contrôle SHA-1 »

REVENIR EN ARRIÈRE (AU NIVEAU « FICHIER »)

QUATRE (OUI, 4 !) COMMANDES DIFFÉRENTES PERMETTENT D'ANNULER DES MODIFICATIONS : **reset**, **revert**, **checkout** ET **restore** (*)

COMMENÇONS PAR CELLES QUI CONCERNENT UN SEUL FICHIER :

- SI ON VEUT RETIRER UN FICHIER DE L'INDEX : `git reset -HEAD script.py`

SON STATUT REPASSERA DE STAGED À MODIFIED, MAIS LE CONTENU NE SERA PAS MODIFIÉ. C'EST L'OPPOSÉ DE `git add`

- SI ON VEUT ANNULER LES MODIFICATIONS SUR UN FICHIER, CES DEUX COMMANDES SONT ÉQUIVALENTES :
  - `git checkout -- script.py`
  - `git restore script.py`

LE STATUT DU FICHIER REPASSERA À COMMITTED, ET SON CONTENU SERA RAMENÉ À L'ÉTAT DU DERNIER COMMIT.
BIEN RÉFLÉCHIR AVANT D'UTILISER CES COMMANDES ! (**)

restore A ÉTÉ INTRODUIT POUR CET USAGE SPÉCIFIQUE, ALORS QUE **checkout** EST UNE COMMANDE PLUS GÉNÉRALE. D'UN POINT DE VUE PÉDAGOGIQUE, **restore** EST DONC PRÉFÉRABLE.

(*) depuis la version 2.23 de git seulement, sortie en août 2019

(**) l'équivalent d'un ctrl-Z, mais le ctrl-Y n'existe pas …

REVENIR EN ARRIÈRE (AU NIVEAU « COMMIT »)

ON ENCHAINE AVEC LES MODIFICATIONS QUI IMPACTENT L'HISTORIQUE DES COMMITS ET/OU LE CONTENU DE L'ESPACE DE TRAVAIL . **revert** PERMET D'INVERSER UN COMMIT QUI VIENT D'ÊTRE RÉALISÉ, EN AJOUTANT UN « CONTRE-COMMIT » DANS L'HISTORIQUE. C'EST LA SOLUTION LA PLUS « SAFE », PRÉFÉRABLE EN TRAVAIL COLLABORATIF(*), MAIS ELLE REND LE LOG PLUS COMPLEXE.

```
git revert HEAD
```

UN PEU PLUS COMPLIQUÉ, **reset** (**) DONT LA PORTÉE DÉPEND DES OPTIONS QUI L'ACCOMPAGNENT :

git reset --soft HEAD~1 : ANNULE LE DERNIER COMMIT, MAIS NE TOUCHE PAS AUX FICHIERS
git reset --mixed HEAD~1 : ANNULE ADD ET COMMIT, MAIS NE TOUCHE PAS AUX FICHIERS
git reset --hard HEAD~1 : RAMÈNE A L'ÉTAT DU COMMIT PRÉCÉDENT, DONC MODIFIE LES FICHIERS …

NB : **git reset HEAD~n** PERMET DE REVENIR EN ARRIÈRE DE N COMMITS (CELA PEUT NOTAMMENT ÊTRE UTILISÉ POUR NETTOYER L'HISTORIQUE)

ENFIN **checkout** SUR UN COMMIT EST UN PEU PARTICULIER : IL PERMET DE S'Y POSITIONNER EN TANT QUE « SPECTATEUR », MAIS LE POINTEUR HEAD SE RETROUVE DÉTACHÉ. JE PRÉFÈRE DISCUTER DE SON USAGE PLUS CLASSIQUE SUR LES BRANCHES À LA PAGE SUIVANTE

(*) car elle ne fait pas disparaitre de commits sur lesquels d'autres auraient pu travailler

(**) dont on a vu page précédente l'utilisation sur un fichier seul

Les branches

Les branches sont surtout utiles dans un contexte collaboratif, puisqu'elles permettent d'isoler son travail du développement principal (il existe ensuite des fonctions pour fusionner les modifications). Mais ca peut évidemment être utile pour un développeur solo, pour tester plusieurs « pistes » …

`git branch maNouvelleBranche` pour créer une branche

`git checkout maNouvelleBranche` pour basculer sur cette branche

`git checkout master` pour revenir sur la branche principale(*)

`git branch -d maBrancheRatee` pour supprimer une branche.

`git branch` pour lister les branches existantes.

Ce type de workflow est utile si vous vous aventurez dans une voie hasardeuse : créer une branche et basculer dessus, continuer à enchaîner les commits, et la supprimer si ça n'aboutit pas.

Comment savoir sur quelle branche on se trouve ? C'est le pointeur nommé HEAD qui le dit. Revenons sur la fin de la sortie de **`git log`**, on voit que HEAD pointe sur la branche principale, nommée master

```
commit e408989122ab6e74738f1e17dad370f5b79c47f7 (HEAD -> master)
```

(*) qui n'a en fait pas de statut particulier. C'est une branche comme une autre, mais c'est la 1e créée, et elle porte toujours ce nom

En vrac

Git, comme Mercurial et Bazaar, sont des solutions décentralisées. Subversion (SVN), solution également populaire, est une solution centralisée. Outre le caractère « politique » de la différence, il est souvent dit que SVN est plus simple d'accès pour le débutant … mais Git est plus à la mode ☺

Github est un service en ligne d'hébergement de dépôts Git

Plus tard, quand vous serez grand :

- étudiez la différence entre **`git merge`** et **`git rebase`** pour fusionner des branches
- étudiez la différence entre **`git pull`** et **`git fetch`** pour récupérer les données d'un repository distant (indice : pull = fetch + merge)

En français, quelques bons sites :

- https://www.atlassian.com/fr/git/tutorials/what-is-version-control
- https://www.pierre-giraud.com/git-github-apprendre-cours/
- https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/intro-git/

Sans oublier les ressources externes listées sur le site officiel : https://git-scm.com/doc/ext

Dernier conseil : évitez de déplacer, renommer ou supprimer des fichiers à la main (utilisez plutôt **`git mv`** et **`git rm`**)