

# Formation Git - TP

SLF

22 octobre 2019

## 1 Introduction

Cette formation a pour but d'être une introduction à git. Elle est assez guidée, mais n'hésitez pas à vous référer aux slides de la présentation et à expérimenter par vous-mêmes !

## 2 Installation

### 2.1 Linux

Vous n'avez très probablement rien à faire du tout, git étant installé par défaut sur les distributions majeures ! Ouvrez un terminal et lancez `git --version` pour vérifier que ça roule.

### 2.2 Windows

Nous recommandons vivement l'utilisation de GitBash.

### 2.3 Configuration

Il y a deux commandes à lancer avant de pouvoir utiliser git, pour que vous soyez correctement affiché comme auteur des commits.

```
git config --global user.name "Jean-Claude Van Damme"
git config --global user.email "jcvd@viarezo.fr"
```

## 3 Le gitlab

Durant cette formation, nous utiliserons le gitlab de ViaRézo (disponible à cette adresse). Vous pouvez vous y connecter en utilisant vos identifiants ViaRézo. Pour pouvoir y accéder via git, il est préférable d'ajouter une clé SSH. Vous pouvez également utiliser l'authentification avec votre compte VR si vous n'êtes pas serein, dans ce cas suivez le lien avec https juste en dessous (il vous demandera vos identifiants). Mais c'est moins bien car il va vous le redemander très régulièrement. Pour l'authentification SSH, il faut ajouter votre clé publique dans les settings (vous pouvez facilement trouver un tutoriel sur internet).

## 4 Premiers pas avec git

### 4.1 Préparation

Pour commencer, clonons un projet réalisé spécialement pour cette formation. Si vous avez utilisé une clé SSH :

```
git clone git@gitlab.viarezo.fr:2018sainratt/formation-git.git
```

Sinon :

```
git clone https://gitlab.viarezo.fr/2018sainratt/formation-git.git
```

Il est également possible de visualiser le projet directement depuis l'interface web du gitlab en cliquant [ici](#). Vous pouvez y voir directement l'arborescence des fichiers et tout un tas d'informations que nous verrons de plus près un peu plus tard. Vous pouvez vérifier directement depuis la ligne de commande que le dossier a bien été cloné en vous déplaçant dans le dossier (`cd formation-git`) puis en listant les fichiers avec (`ls`).

## 4.2 Premier commit

Visualisons tout d'abord l'état de notre projet. Si vous avez oublié la commande, retournez voir les slides ! Maintenant, modifions un des fichiers. Choisissez en un et ouvrez votre éditeur de texte préféré pour le modifier (n'oubliez pas d'enregistrer après). En visualisant à nouveau l'état du projet, la modification est apparue ! Cependant, elle n'a pas encore été sélectionnée pour être ajoutée dans le prochain commit... On utilise donc la commande

```
git add <nom du fichier>
```

Vous devriez observer que la modification apparaît maintenant en vert et est prête à être commit. Il est donc l'heure de faire notre premier commit !

```
git commit -m "<un message sympa>"
```

Si vous regardez à nouveau le status, vous ne verrez plus le fichier : il fait maintenant partie intégrante du code !

## 4.3 Revenir en arrière (quand tout est cassé)

Petit hic cependant... nous n'avons pas créé de branche au préalable, ce qui signifie que notre commit se trouve directement sur la branche master ! Dans la vraie vie, cela voudrait dire que notre code se retrouverait sur le site en production à la prochaine mise à jour, possiblement sans avoir été relu !

Heureusement, il est toujours possible de revenir en arrière avec git. Pour cela, la commande à utiliser est

```
git reset --hard <commit>
```

Cette commande permet de faire revenir la branche à un commit antérieur (il y a d'autres modes que `--hard` mais ce n'est pas le sujet). Il nous suffit donc de récupérer le commit précédent et utiliser la commande. Pour cela, deux méthodes : soit on utilise le fait qu'il soit tout simplement le commit précédent en l'appelant par

```
HEAD^ (ou HEAD~1, HEAD~2 étant le grand parent et ainsi de suite...)
```

soit on dénomme le commit par son hash. Pour récupérer ce hash, il nous suffit de récupérer l'historique des commits avec la commande

```
git log
```

qui affiche l'historique des commits dans la branche du plus jeune au plus vieux. Le hash est la longue chaîne de caractère qui désigne le commit. En pratique, on peut presque tout le temps se limiter aux 8 premiers chiffres du hash, il suffit que tous les commits en aient des différents (et c'est normalement le cas).

Revenons à nos moutons. Ayant effectué le `git reset`, nous sommes revenus à la situation de départ (vous pouvez vérifier). Déterminés à faire proprement, nous allons pouvoir nous atteler à la création de branches !

## 5 Un monde de branches

Pour l'instant, nous sommes restés sur notre unique branche : **master**. Rappelons que les branches sont simplement des pointeurs vers un commit, nous allons illustrer ça très vite. Pour cela, créons déjà notre première branche avec

```
git branch <nom de la branche>
```

A première vue, rien n'a changé. mais si vous faites `git branch` (sans rien devant), ce qui affiche la liste des branches, vous devriez voir votre nouvelle branche apparaître. Un coup d'oeil à `git log` permettra même de voir qu'elle pointe sur le même commit que **master**. Mais alors, comment change-t-on de branche ? On utilise

```
git checkout <nom de la branche>
```

Vous devriez alors voir le nom de la branche changer dans votre terminal (cependant, comme elle pointe sur le même commit que master, rien n'a changé dans les fichiers). Faites donc un commit sur cette branche. Maintenant, si on utilise `git log`, on verra que la nouvelle branche pointe sur le nouveau commit, tandis que **master** n'a pas bougé de l'ancien. Si on rechange de branche pour aller sur **master**, on reviendra donc à l'état initial. Et si on commit maintenant sur master, cela n'affectera pas l'autre branche (faites-le!).

## 5.1 La fusion, sujet sensible

Encouragé par votre feuille de TP, vous avez donc maintenant deux branches qui ont divergé. Mettons maintenant que vous voulez incorporer les changements de votre nouvelle branche à la branche principale (ici **master**, mais on pourrait faire une branche partant de n'importe quelle autre branche). Placez vous bien sur la branche **master** avant de continuer. La commande à utiliser est naturellement

```
git merge <branche>
```

Git va alors créer un commit avec deux parents (les deux commits des branches respectives), contenant les modifications des deux branches, et faire pointer master dessus (l'autre branche n'a pas bougé par défaut). Si vous n'avez pas modifié les mêmes zones des fichiers dans les deux branches, tout ça est fait automatiquement (comme par magie). Sinon, vous aurez un message qui vous dit qu'il y a des conflits. Dans cette situation : soit vous faites

```
git merge --abort
```

pour annuler la fusion, soit vous réglez les conflits et faites

```
git merge --continue
```

ce qui créera le commit de merge comme dans le cas sans conflits. Mais comment règle-t-on ces conflits ?

Allez voir dans un des fichiers qui est indiqué comme posant un conflit. Vous verrez des marqueurs indiquant les deux versions des lignes incriminées. Editez le fichier pour garder la version que vous voulez (ou un mix des deux), puis terminez la fusion.

## 6 Du côté du Gitlab...

Il est l'heure de voir un peu comment ça se passe sur l'interface Web du Gitlab. Rendez vous ici à nouveau. Les fichiers que vous voyez actuellement sont ceux de la branche master, que vous avez récupérés au début du TP. Il est possible de changer la branche en ouvrant le petit menu déroulant prévu à cet effet (au dessus du nom et de la description du dernier commit en date).

Dans les onglets à gauche, vous pouvez trouver tout un tas d'informations. On ne va s'intéresser qu'aux 4 premiers onglets (et c'est déjà pas mal!) :

- Le premier permet essentiellement de revenir sur la page d'accueil du projet (où vous êtes déjà donc ne faites pas ça maintenant)
- Le deuxième permet d'accéder aux listes des différentes branches, des commits, des contributeurs...
- Le troisième concerne les issues. Il s'agit d'un outil très pratique pour gérer les projets : vous pouvez créer une issue quand il faut régler un bug ou ajouter une nouvelle feature, l'assigner à quelqu'un, la relier à des merge requests... cela permet de garder une trace de ce qu'il y a à faire et de marquer quand c'est résolu.
- Le quatrième concerne les merge requests. Les merge requests permettent de fusionner des branches directement sur le Gitlab, ce qui est beaucoup mieux que de faire les fusions en local : on a une trace de ce qui a été fait, et les autres gens peuvent relire le travail effectué et le valider (ou non), ou même poster des commentaires.

Je vous conseille fortement de jeter un oeil aux différents onglets pour vous familiariser.

Il est maintenant temps d'envoyer vos changements sur le Gitlab. Vous pouvez utiliser **git push** comme vu dans la formation (attention, il y a une petite subtilité!). Ensuite, vous pouvez récupérer la branche de quelqu'un d'autre en faisant **git pull** et en changeant de branche. Essayez de créer votre propre merge request sur le Gitlab !

## 7 The advanced stuff (and random tips)

A partir de ce que vous avez fait jusque là, vous pouvez utiliser git dans vos projets. La suite, c'est surtout pour vous aider à l'utiliser correctement et proprement (et ça peut rester très utile). Retournez sur votre branche histoire de pas faire n'importe quoi (ou faites n'importe quoi si vous voulez, ça ajoute de l'entropie dans ce TP et c'est formateur)

Directement un petit tip pour vous aider à garder un historique clean : si vous faites **git commit --amend**, les changements que vous avez add seront rajoutés dans le dernier commit que vous avez fait ! Testez donc (pour sortir, faites **:wq**, ne vous posez pas de questions) ! Ensuite, essayez de push vos changements. Selon toute probabilité, Git refuse de push car votre commit est différent de celui qu'il a déjà. Il faut donc le forcer et passer outre son consentement (ne faites pas ça chez vous, enfin pas en dehors de git) et faire un **git push --force**.

ATTENTION, n'abusez pas de cette commande!!! Dans le cas d'un **--amend** ça va, mais dans d'autres cas vous risquez de supprimer un tas de changements qui ont été faits sur la branche, car vous réécrivez l'historique. Utilisez la donc avec précaution. Ceci dit, l'historique est plus clean et ça c'est bien.

## 7.1 Rebase

Un autre bon plan pour avoir un historique clean, c'est le rebase. Le rebase consiste à rejouer vos commits sur la branche source plutôt que de créer un commit de merge et effectuer la fusion. Avantage : il n'y a pas de commit de fusion, on pourrait avoir l'impression que tous les commits ont été faits sur master (sauf que les dates sont un peu aberrantes). Pour faire un rebase, il vous suffit de faire `git rebase <branche source>` (souvent `git rebase master`) quand vous êtes sur la branche, et les commits de master seront "rajoutés" en dessous de vos commits. Vous pouvez ensuite merge et il ne créera pas de nouveaux commits (on parle de "fast-forward"). Je vous engage à tester en faisant des commits sur master puis en rebasant.

## 7.2 `git rm` et `git mv`

Ces commandes vous permettent de retirer ou renommer ou déplacer des fichiers. Vous pouvez facilement trouver comment ça marche...

## 7.3 `git diff`

Cette commande sert à voir les lignes ajoutées ou retirées entre deux commits, ou entre votre version et le commit actuel (ou autre...). Vous pouvez regarder la documentation de cette commande également, elle peut s'avérer très utile.

## 7.4 Le `.gitignore`

Si vous regardez bien, vous verrez que dans le dossier de votre projet il y a un fichier appelé `.gitignore`. Ce fichier permet de dire à Git de ne pas se préoccuper de certains fichiers (ils n'apparaîtront donc pas lors d'un `git status` par exemple). Il suffit de mettre le chemin relatif des fichiers que vous voulez ignorer.

# 8 Conclusion

Dans ce TP vous avez normalement vu tout ce dont vous avez besoin pour utiliser Git de manière sereine.