TD COLD #2 - Partie 2

# GIT

## GESTION DE VERSION

*Git* est un logiciel essentiel pour les informaticiens : il permet non seulement de gérer différentes versions d'un projet sous forme d'une hiérarchie de documents (généralement du code), mais aussi de partager ces versions et de travailler de manière collaborative sur leur contenu.

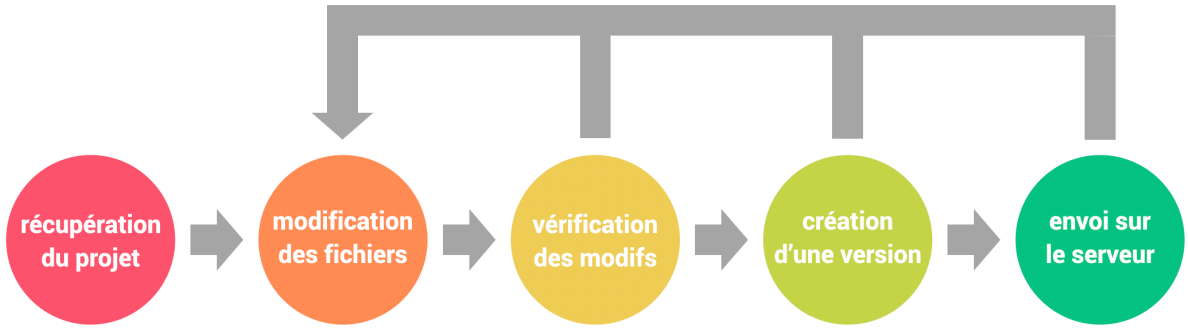
Afin de faciliter l'apprentissage de ce logiciel qui peut paraitre complexe, nous allons aborder ses différentes fonctionnalités au fur et à mesure. Pour l'instant, nous nous concentrerons sur la partie gestion de version et (un peu) partage.

### Versions

Une version dans *git* s'appelle un **commit**. Un *commit* contient une version du projet, c'est à dire une version d'un répertoire et de son contenu. Au fur et à mesure du temps, plusieurs versions du projet vont être créées, chacune contenant peu ou beaucoup de modifications par rapport à la version précédente.
Le but de *git* est de manipuler ces *commits* : les créer, les gérer, naviguer d'une version à une autre, voire les envoyer sur un serveur pour y être partagées avec d'autres développeurs.

### Etapes

Les étapes standards lors du travail sur un projet sont les suivantes :




*Git* va être utilisé dans chacune de ces étapes (à part la modification des fichiers).

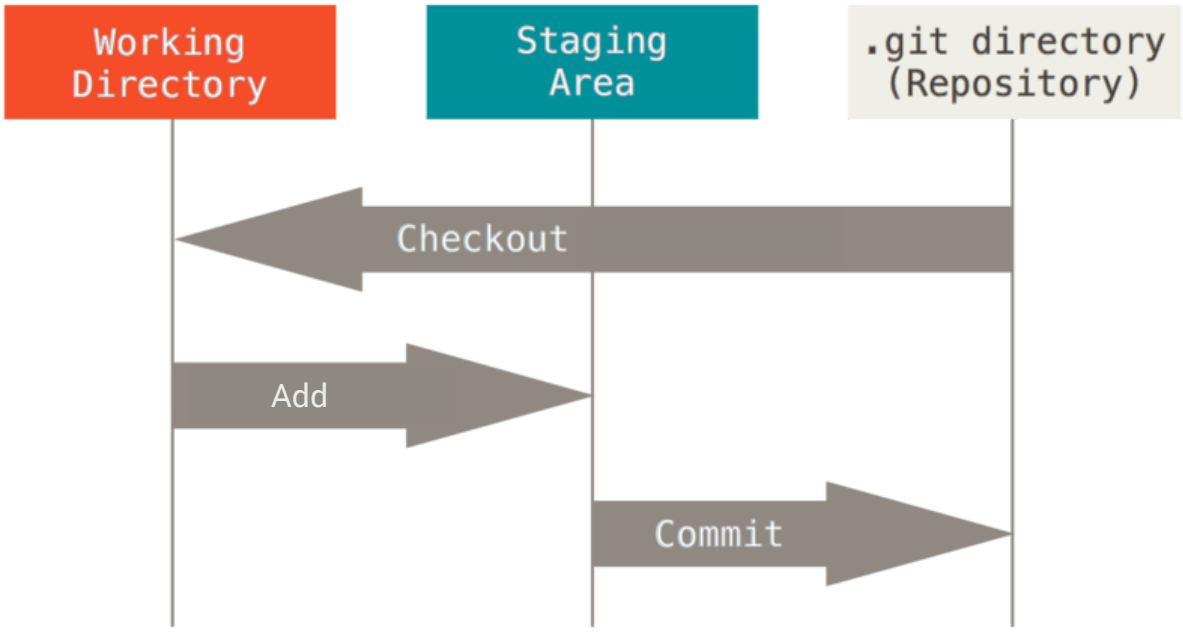
## Zones

Pour *git*, un fichier (ou une modification d'un fichier) peut se trouver dans l'une des 3 zones suivantes (ces 3 zones sont sur le disque dur local) :

- **working directory** (répertoire de travail) : l'endroit normal, là ou sont les fichiers sur lesquels le développeur travaille ;
- **repository** (répertoire .git) : là ou sont stockées toutes les différentes versions créées (les *commits*). Il se trouve dans le répertoire *.git* à la racine du projet ;
- **staging area** (stage, ou index) : zone spéciale qui retient les modifications avant d'en faire une nouvelle version.

Le *stage* (ou i*ndex*) sert dans les utilisations avancées de *git* à gérer finement la création des *commits*. Dans l'utilisation que l'on va en faire pour l'instant, il ne sera d'aucune utilité, mais *git* obligeant son utilisation, il faudra quand même le prendre en compte.

Le passage d'une zone à une autre se fait suivant le schéma suivant :



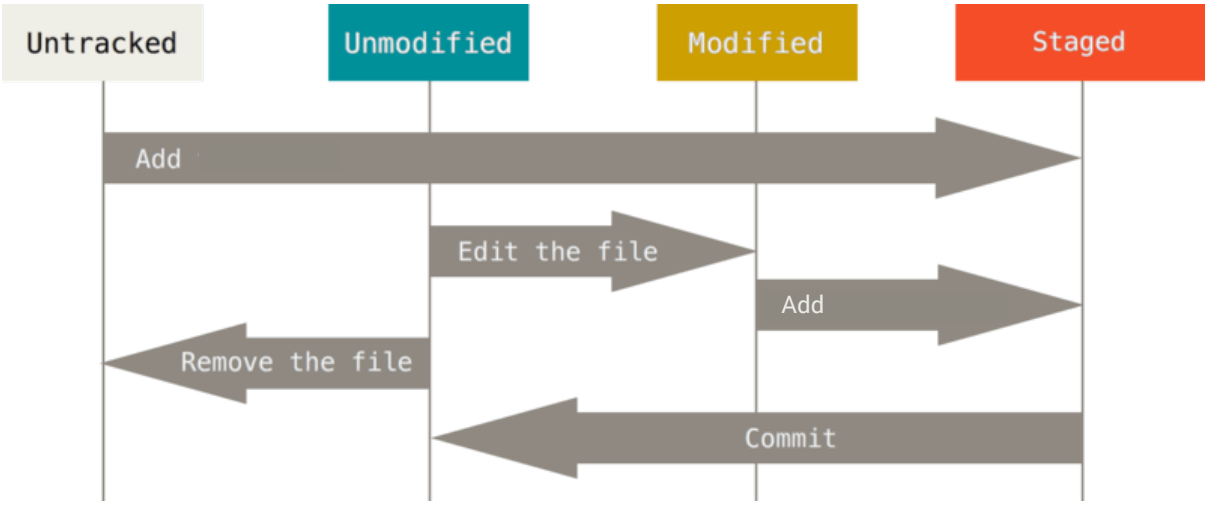

Source : https://git-scm.com

## Modification des fichiers

*Git* se base sur l'état des fichiers pour savoir quelle action appliquer. Pour *git*, un fichier peut se trouver dans un des 4 états suivants :

- **untracked** : *git* ne s'occupe pas du fichier ;
- **unmodified** : le fichier n'a pas changé depuis la dernière version ;
- **modified** : le fichier a été modifié mais n'a pas encore été mis dans l'index ;
- **staged** : le fichier a été modifié et mis dans l'index.

Le passage d'une zone à une autre se fait suivant le schéma suivant :




Source : https://git-scm.com

## .gitignore

Parfois, certains fichiers ne doivent jamais être pris en compte par *git* (par exemple les fichiers binaires ou les fichiers temporaires résultant de la compilation). Dans ce cas, pour éviter de les ajouter à *git* par erreur, et pour éviter de polluer les logs, il est possible de dire à *git* d'ignorer ces fichiers, simplement en mettant leurs noms dans un fichier .**gitignore**, placé à la racine du projet.

# OUTILS

Vous trouverez sur Arche trois sites très utiles :

- *Git Flight rules* : un livre de recette git, classant, par type d'action à faire, quelle commande utiliser.
- *Git command explorer* : la même chose mais plus interactive.

- *.gitignore file creator* : un site permettant de générer automatiquement un fichier .gitignore.

## COMMANDES

Voici une liste des commandes que vous devez connaitre, avec les principaux attributs utiles (mais il en existe pleins d'autres, à vous de vous renseigner). N'oubliez pas de consulter la documentation.

### git config —global <config.name> <config.value>

Configure le nom et l'adresse email mis automatiquement dans les *commits* par *git*. Il faut notamment configurer dès le début *user.name* et *user.email*, si cela n'a pas déjà été fait. C'est une bonne idée de configurer aussi *core.editor* sur la valeur « nano ».

### git clone <url>

Clone le projet de l'url donné sur le disque local. Récupère non seulement la dernière version mais aussi toutes les anciennes versions.

### git status

Affiche pour chacun des fichiers du projet son état, parmi les 4 possibles. Très utile pour savoir quels fichiers ont été modifiés, lesquels ont été mis dans l'index, et lesquels restent en dehors de l'index.

### git log

Affiche la liste des versions de ce projet, en commençant par la dernière. Utile pour voir si le dernier commit a bien été pris en compte.

### git diff <nom_du_fichier>

Affiche les modifications du fichier donné depuis la dernière version. Très utile pour revoir toutes les modifications qui feront parties de la prochaine version.

### git add <nom_du_fichier>

Ajoute le fichier en question dans l'index. Ne fait rien si le fichier n'a pas été modifié. A noter, on peut aussi passer le chemin d'un répertoire, et dans ce cas tous les fichiers du répertoires sont ajoutés à l'index.

Comme pour l'instant l'index n'est pas utilisé, il est plus pratique de faire directement « *git add .* » depuis la racine du projet.

## git commit -m « <message> »

Crée une nouvelle version à partir des modifications mises dans l'index. Les modifications qui ne sont pas dans l'index n'apparaitront pas dans le commit. Attention à ne pas oublier l'option -m suivi du message sinon git ouvre un éditeur de texte en ligne (normalement « nano » si vous l'avez bien configuré) pour entrer le message.

## git push

Envoie sur le serveur tous les commits qui ne sont pas déjà sur le serveur.

## git pull

Récupère en local tous les commits qui sont sur le serveur mais pas encore en local.