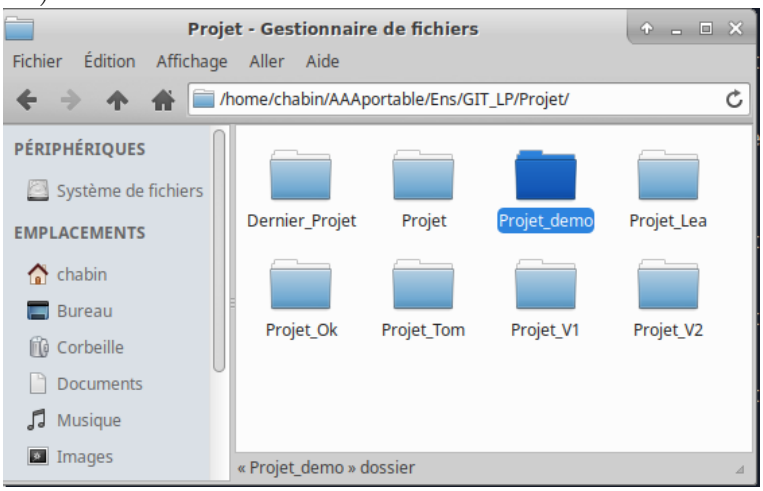
UE1 cons : Consolidation Bases de Données et Programmation Objet.
Versionnage et Tests.

Cours-TD n°1

Introduction à git

git est un logiciel de gestion de versions décentralisé. C'est un logiciel libre créé par Linus Torvalds, auteur du noyau Linux, et distribué selon les termes de la licence publique générale GNU version 2. (Source : WIKIPEDIA)



Que contiennent tous ces répertoires ? Quelle est la version à rendre qui fonctionne ? Comment faire pour que mes dernières modifications soient aussi aussi prises en compte dans ma version de démo ? Comment faire pour fusionner le travail de Tom et Léa ? Comment récupérer le travail que j'ai effectué hier soir sur ma machine personnelle ? git va nous permettre de répondre à toutes ces questions (et à bien d'autres que l'on ne verra pas dans cette introduction.)

Cette première feuille va aborder l'utilisation de git en local comme outil de gestion de version.

1. Paramétrage et initialisation. Pour pouvoir utiliser git, on doit configurer auparavant les paramètres suivants :

```
git config --global user.name "Jane Abi"
git config --global user.email "jane.abi@etu.univ-orleans.fr"
```

2. Obtenir de l'aide. Le site officiel de git est **https ://git-scm.com/**. La commande principale est **git** suivi du nom de la commande git que l'on veut exécuter et éventuellement d'arguments.

```
git --help  // liste des commandes les plus utilisées.
git help -a // toutes les commandes de git
git help init // aide sur la commande init
```

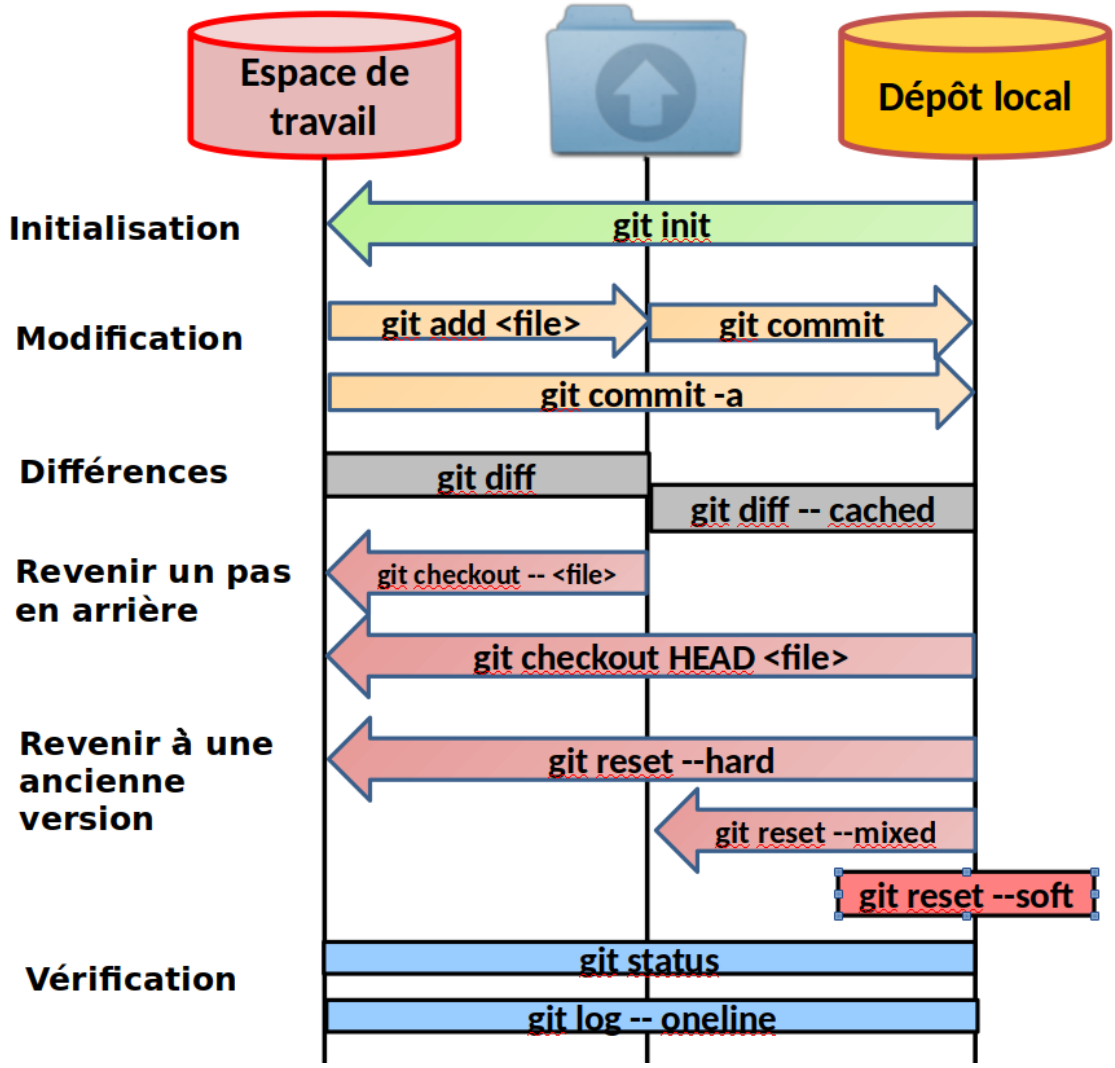
3. Initialisation d'un projet vide.

```
git init monPremierProjet
```

   Que constatez-vous ?
   Quel est l'état de notre projet ?

```
git status
```

Avant de poursuivre regardons la structure générale qui est en place :



Il y a trois zones distinctes qui contiennent votre travail :

l'espace de travail qui contient les fichiers eux-mêmes sur votre disque dur, que vous éditez grâce à votre éditeur préféré. C'est votre répertoire de travail, ou Working Directory. Il faut penser à la copie de travail comme un bac à sable où vous pouvez essayer vos modifications avant de les transférer dans votre index puis le valider dans votre historique.

Ensuite, une mystérieuse zone spéciale que l'on appelle l'index, ou la zone de staging, elle va contenir une image de la version que l'on veut conserver.

Enfin, la base de données de Git où est stockée l'arborescence de votre projet telle qu'elle était lors de votre dernier commit ainsi que toutes les versions précédentes ; c'est le sous répertoire **.git**.

4. Création d'une première version.
   (a) Créez un fichier **rapport.txt** contenant la ligne "Partie 1 de nom rapport" puis tapez : `git status`.
   (b) Placez ce fichier dans l'index pour qu'il soit versionné :

```
git add rapport.txt
```

Tapez : `git status`.

(c) Créez une version de votre rapport actuel dans le dépôt local :

```
git commit -m "Premiere partie"
```

(d) Vérifiez l'état de votre projet : `git status`. La "copie de travail est propre" ce qui signifie que les trois zones contiennent (pointent) vers les mêmes fichiers de votre projet. Vous avez une première version de votre travail.

5. Création d'une seconde version.

(a) Modifiez le fichier **rapport.txt** en ajoutant une seconde ligne la ligne "Partie 2 de nom rapport" puis testez votre projet.

```
git status
git diff
git diff --cached
```

Commentez chaque commande.

(b) Placez ce fichier dans l'index pour qu'il soit versionné puis testez votre projet.

```
git add rapport.txt
git status
git diff
git diff --cached
```

Commentez chaque commande.

(c) Créez une nouvelle version de votre rapport actuel dans le dépôt local :

```
git commit -m "Seconde partie"
```

Vous pouvez faire les deux dernières étapes en une seule avec la commande :

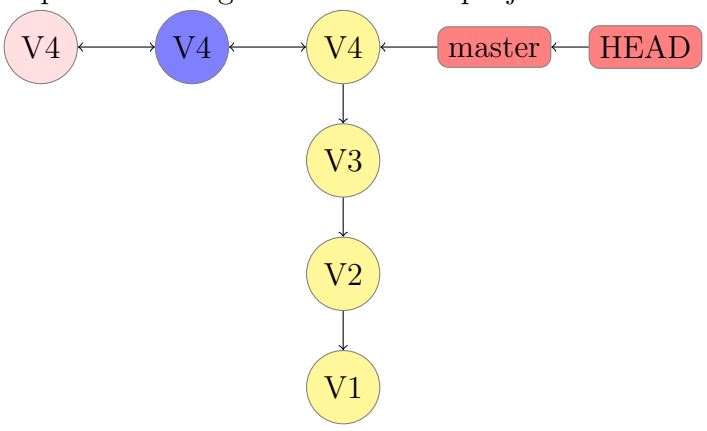
```
git commit -am "Seconde partie"
```

Cette commande va placer automatiquement dans la "staging zone" tous les fichiers de votre répertoire de travail qui sont suivis (vous avez déjà fait un git add de ces fichiers auparavant) et créer une version dans le dépôt local.

(d) Vérifiez l'état de votre projet : `git status`. La "copie de travail est propre" ce qui signifie que les trois zones contiennent (pointent) vers les mêmes fichiers de votre projet. Vous avez deux versions de votre travail. Pour voir ces versions :

```
git log
git log --oneline
```

6. Créez une version 3 puis une version 4 de votre rapport.
7. Représentation générale de votre projet :



master est le nom de la branche principale de votre dépôt, il pointe en général sur la dernière version (le dernier commit) conservée dans cette branche.
HEAD est un pointeur sur la référence de la branche actuelle. Ceci signifie que HEAD sera le parent du prochain commit à créer.
Chaque version pointe sur sa version parente.
A chaque version (commit) est associé un code, un auteur et une date :

```
commit 3f63ab1302a8816039b880bfbe273f3b2c52d86c
Author: Jacques Chabin <jacques.chabin@univ-orleans.fr>
Date:   Tue Jul 17 10:42:19 2018 +0200
```

Pour identifier un commit on pourra, lorsqu'il n'y aura pas ambiguïté, utiliser que les 4 premiers caractères.

8. Voyager dans les différentes versions. Ensemble de vos versions :

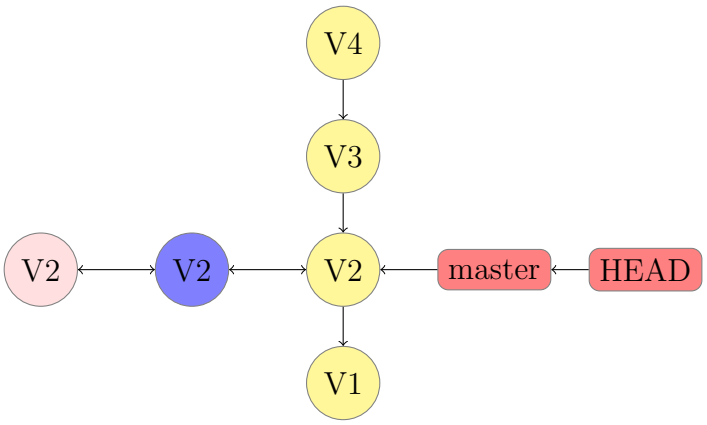
```
git log --oneline  donne comme résultat

fc2c3a6 (HEAD -> master) Partie 4 finie
bcbfed6 Partie 3 finie
06cd67d Seconde partie
3f63ab1 Première partie
```

Cela correspond au schéma de l'item 7
Tapez la commande

```
git reset --hard 06cd
```

Vous arrivez sur cette représentation :



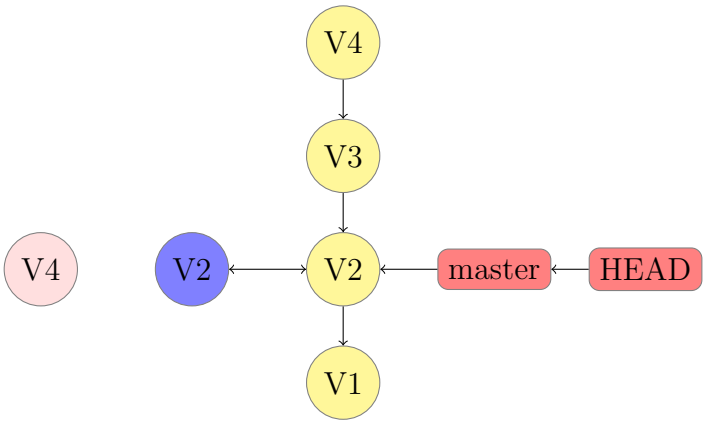
La commande `git reset --hard 06cd` déplace **HEAD** et récupère dans l'index et le répertoire de travail, les fichiers de la version V2. C'est comme si vous aviez supprimé vos deux derniers commits (deux dernière version de votre projet). Regardez sous un éditeur l'état de votre fichier **rapport.txt** il correspond au fichier de votre seconde version. Attention en observant la représentation on a l'impression d'avoir perdu V3 et V4 (plus de pointeur sur ces commits). On peut les retrouver en ayant conservé le numéro du commit V4 et et tapant `git reset --hard fc2c` on revient au schéma de l'item 7.
On peut remplacer l'identifiant du commit que l'on veut atteindre par **HEAD~** ou **HEAD~2** pour remonter d'une version ou de deux versions (ou de x versions). En partant de la représentation de l'item 7, `git reset --hard HEAD~2` fait la même chose que `git reset --hard 06cd`.
Si vous voulez supprimer des commit, il suffit de repartir du commit que vous avez atteint avec le reset et de refaire des commits.

L'option `--hard` met dans l'index et dans le répertoire de travail les fichiers correspondant à la version atteinte. On peut remplacer cette option par `--mixed` (valeur par défaut) ou `--soft`.

En partant de la représentation de l'item 7

```
git reset --mixed 06cd
```

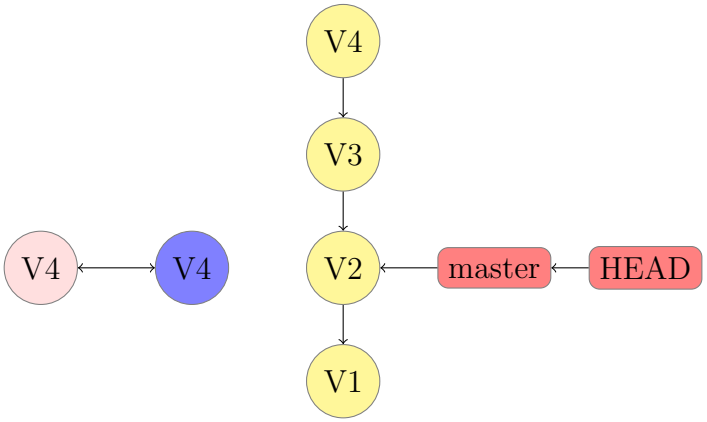
déplace **HEAD** et récupère dans l'index l'état des fichiers de la version V2. Le répertoire de travail n'est pas changé, il contient toujours les fichiers de V4.



En partant de la représentation de l'item 7

```
git reset --soft 06cd
```

Déplace **HEAD** sans modifier l'index et le répertoire de travail.



9. Une autre utilisation du reset. En partant de la représentation de l'item 7

```
git reset  06cd -- rapport.txt
```

Va placer dans l'index le fichier rapport.txt qui correspond au commit 06cd. On peut ensuite le récupérer dans le répertoire de travail (en écrasant l'ancien fichier) en faisant `git checkout -- rapport.txt` (On ne peut pas utiliser l'option –hard sur un reset qui utilise un chemin.)

10. Réparer une erreur sur un nouveau commit.

On peut créer un nouveau commit qui annule les changements du dernier commit.

```
git revert HEAD
```

Cela crée un nouveau commit qui annule les changements dans HEAD. Vous devrez mettre le message de ce nouveau commit.

11. Remarque si votre projet contient des fichiers sources et des fichiers compilés, on ne conserve dans le dépôt git que les fichiers sources. On peut créer un fichier nommé **.gitignore** contenant le nom des fichiers (ou des répertoires) que l'on ne veut pas suivre (que l'on ne veut pas versionner, un nom par ligne. En général ce fichier est suivi `git add .gitignore`. Exemple de fichier **.gitignore**

```
__pycache__
*~
test_pointbis.py
```