0 INTRODUCTION AU VERSIONING AVEC GIT

AYEDJO Kokou Wisdom, DOMLAN Karl Jason Akoété

September 10, 2025

Contents

Ce chapitre n'a pas de lien direct avec la POO ou java. Cependant les codes que nous aurons à écrire et les exercices et projets qui seront soumis seront hébergés sur Github. Il est donc important d'avoir des notions de versionnage avec *git*. Elle vous seront bien utiles, au-delà même de ces cours.

C'est quoi le versionnage (versioning)?

Lorsque vous écrivez un programme, vous le sauvegardez avant de l'exécuter. Et à chaque modification vous faites pareil avant de le tester. Le problème est qu'à chaque nouvelle sauvegarde que vous faites, la version précédente est écrasée. Si vous êtes un développeur qui aime expérimenter, tester et essayer de nouvelles choses avec son code (jusqu'à ce qu'il ne marche plus du tout, bien souvent), vous voudrez très certainement garder les versions antérieures de vos projets.

Les **Systèmes de Contrôle de Version (VCS)** servent justement à cela. Ce sont des systèmes qui enregistrent les changements dans un fichier ou un ensemble de fichiers au cours du temps afin de pouvoir **rapeller** des versions spécifiques plus tard. Les VCS permettent de ramener un projet entier à un état antérieur, savoir qui a modifié quelle partie du projet et quand, etc… Utiliser un VCS signifie qu'on peut facilement revenir à un état fonctionnel quand on fait des erreurs.

Les types de VCS

- **Les VCS en local**: Ils existent depuis longtemps. La base de données qui enregistre les différentes versions des fichiers est stockée en local sur une même machine. L'un des plus populaires s'appelle *RCS*. Leur principal inconvénient est assez évident.
- **Les VCS centralisés**: Tous les fichiers sous contrôle de version sont stockés sur un serveur central. Des exemples en sont: *Subversion* et *Perforce*. C'était le standard pendant plusieurs années. Ce modèle offrait plusieurs avantages sur les VCS en local, notamment la collaboration. Mais l'inconvénient principal est qu'en cas de panne du serveur, tous les travaux seraient inaccessibles.
- **Les VCS distribués**: Avec les VCS distribués, les clients ont une copie intégrale des fichiers sous contrôle de version, ainsi que de tout leur historiques des versions. Ainsi, si un serveur est en panne, il peut être restauré avec n'importe quel dépôt client.

Chaque clone est un back-up de toutes les données. Le plus populaire est git et c'est celui que nous allons étudier.

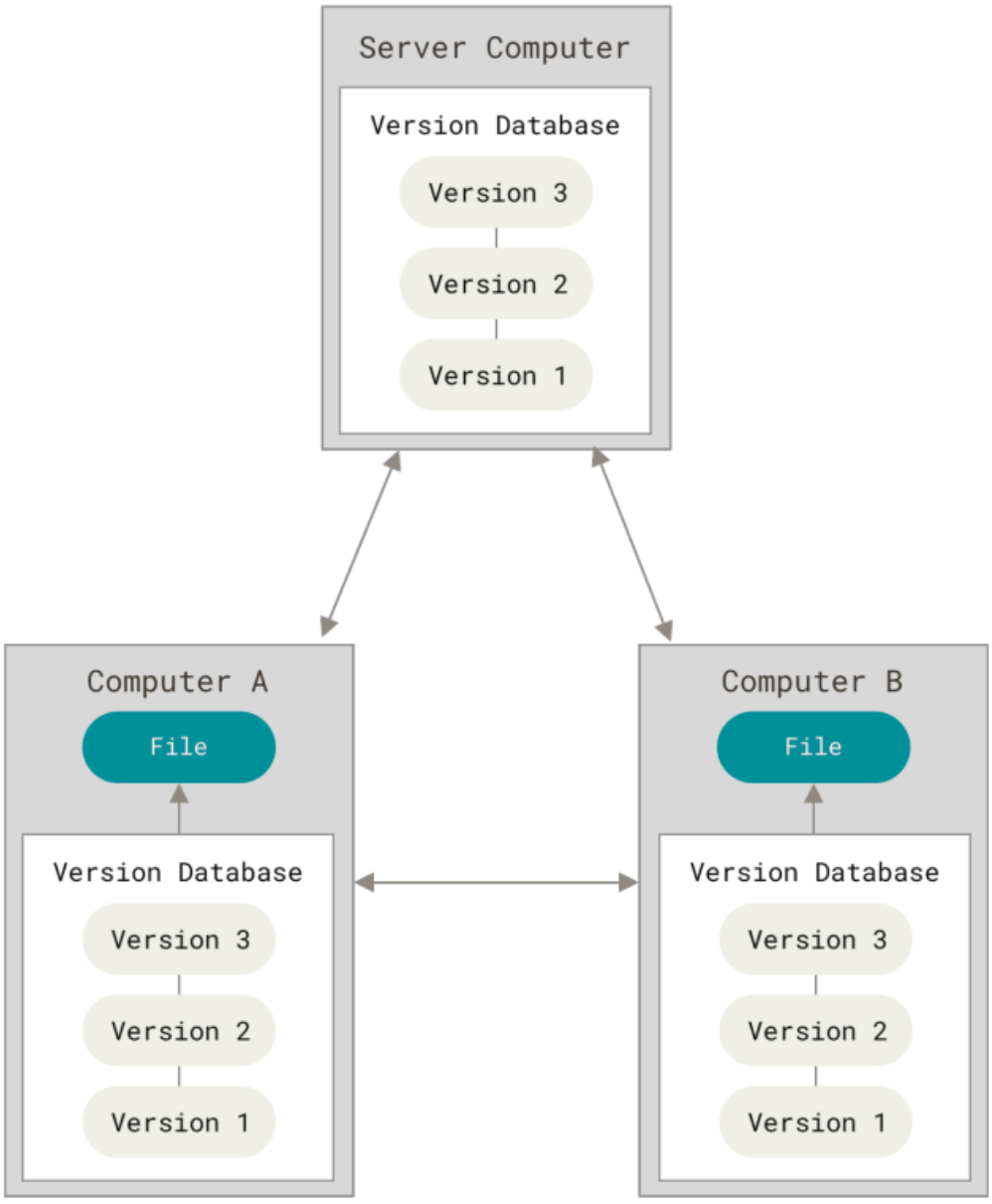

Figure 1: DIAGRAMME D'UN VCS DISTRIBUÉ

NE CONFONDEZ SURTOUT PAS GIT GITHUB ET GITLAB

- git est un VCS (Système de Contrôle de version) qui surveille et enregistre les changements dans un ensemble de fichiers.
- *Github* et *Gitlab* sont des plateformes en lignes qui permettent d'héberger votre code source à distance.

Découvrons Git

Comme dit précédemment, git est un VCS distribué. Il a été créé en 2005 par la communauté de développement de Linux et en particulier par Linus Torvalds. Ses objetifs principaux étaient la rapidité, un design simple, et le support de nombreuses branches parallèles, être complètement distribué et capable de gérer de grands projets.

Les principes de fonctionnement de git

1. **Surveillance basée sur les snapshots**: À chaque sauvegarde, git enregistre une "image" de ce à quoi ressemblent tous les fichiers du projet. Si un fichier n'a pas changé depuis la dernière sauvegarde, git ne l'enregistre pas à nouveau.
2. **presque toutes les opérations sont en local**: Git ne nécessite pas de connexion à un serveur distant pour récupérer les historiques. Il est possible de travailler hors ligne avec git.
3. **git vérifie l'intégrité**: Grâce à un checksum basé sur SHA-1, git calcule les valeurs de hachage des fichiers. En vérité git stocke presque tout avec les valeurs de hachage. Cela garanti que les données stockées ne subissent aucune altération.
4. **Les trois états**: C'est le concept le plus important de git. En effet git perçoit tous les fichiers sous trois états principaux:
    - **Modifié**: signifie que vous avez changé le fichier mais qu'il n'est pas encore enregistré dans la base de données de git.
    - **Staged**: signifie que vous avez marqué un fichier modifié pour être enregistré dans la base de données lors de la prochaine capture. En effet le stage (ou index) est la zone de transit où sont sélectionnés les changements à enregistrer.

- **committed**: signifie que les changements ont bien été stockés dans votre base de données locale.

Le workflow typique avec git est le suivant:

1. Vous modifiez des fichiers dans votre répertoire (dossier ou projet)
2. vous ajoutez les changements que vous voulez enrégistrer à l'index. C'est comme dire à git "je veux enregistrer les modifications que j'ai faites sur ces fichiers" en pointant bien du doigt les fichiers en question.
3. vous faites un `commit`. C'est l'opération qui capture les fichiers tels qu'ils sont dans l'index et les ajoute à la base de donnée.

Toutes ces opérations s'effectueront en général dans un terminal. En effet git est d'abord un outil en ligne de commande (CLI).

Les bases de git

Obtenir un dépot git (repository)

On peut soit le créer soi-même ou cloner un repository existant.

Initialiser un repository

Si vous avez un dossier qui n'est pas encore sous contrôle de version, pour commencer:

- allez dans le répertoire en question

```
$ cd /home/user/my_project
```

- initilisez le repository avec

```
$ git init
```

  Ceci crée un nouveau sous-dossier nommé `.git` qui contient tout ce dont git a besoin. Mais à ce stade aucun fichier n'est encore surveillé par git.

- ajoutez des fichiers à l'index avec

```
$ git add .
```

Le `.` est important ici. Il signifie “tout”. En d’autres termes on dit à git d’ajouter tous les fichiers du repository à l’index. Cela veut dire qu’ils seront tous “capturés” au prochain commit. Il est bien-sûr possible de préciser les fichiers à ajouter.

- enregistrez l’état des fichiers tel qu’ils sont été capturés dans l’index avec:

```
1 $ git commit -m "premier commit"
```

Le paramètre `-m` est le message du commmit. Vous devez décrire un peu ce qui a été modifié avant ce commit. Si le paramètre `-m` n’est pas entré, git ouvre l’editeur de texte par défaut (que vous pouvez choisir avec `git config --global core.editor`)

Repository, dépôt, répertoire, dossier, projet, ces mots peuvent être confus au départ. Mais de façon très grossière, pensez-y comme étant le dossier dans lequel vous avez fait le `git init`.

Cloner un repository existant

Il est possible de cloner un repository existant. Allez dans le dossier où vous vouler l’obtenir et faites:

```
1 $ git clone https://github.com/compte/projet
```

Un nouveau dossier nommé `projet` sera créé avec un sous-dossier `.git` et tout l’historique du repository.

Il est important de vous rappeler que nous travaillons toujours dans une invite de commande. Donc quand nous disons “aller dans un dossier” cela signifie faire `cd chemin/du/dossier`. Vous pouvez copier le chemin depuis votre explorateur de fichiers.

Ici nous avons utilisé une URL `https` mais vous pouvez également utiliser une URL `ssh` (secure shell).

Statut des fichiers surveillés

Vous pouvez voir le statut des fichiers en lançant :

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

cela signifie que tous les fichiers que git surveille sont à jour, n'ont pas été modifiés. Nous verrons ce que sont les branches un peu plus tard.

- Lorsque vous créez nouvellement un fichier, git ne le surveille pas. Il faudra faire un `git add fichier` pour que git commence à le surveiller.
- quand vous initialisez un dépôt, git crée une branche principale qui porte par défaut le nom `master`.
- Lorsque vous faites `git commit`, ce que git ajoute à sa base de données, c'est l'état des fichiers tels qu'ils sont dans l'index (s'ils y sont), c'est à dire ce que vous avez ajouté précédemment avec `git add`. Ce n'est pas leur état actuel dans votre dossier. Le `git add` est donc essentiel pour que git sache ce que vous voulez ajouter dans le prochain commit. Sinon il est possible de faire `git commit -a`. Le paramètre `-a` indique à git qu'il faut automatiquement ajouter à l'index tous les fichiers qui étaient déjà surveillés avant le commit. Cela n'incluera donc pas les nouveaux fichiers nouvellement créés depuis le commit précédent.

Git a beaucoup d'autres commandes et beaucoup d'autres paramètres disponibles avec celles que nous avons déjà parcourues.

Pour aller plus loin

- `git diff` vous montre exactement quelles lignes ont été ajoutées et retirées entre l'index et l'état actuel de votre projet. En d'autre termes la différence entre ce qui est dans l'index et ce qui a été modifié mais pas encore ajouté.
  - `git diff --staged` vous montre les changements qui sont indexés et qui seront dans le prochain commit. Il compare l'index au commit précédent.
- `git log` vous montre un historique des commits précédents

- Le fichier `.gitignore` indique les fichiers et répertoires qu'on ne veut pas que git enregistre, souvent parce qu'ils sont générés automatiquement par le système ou d'autres programmes. Nous ne les couvrirons pas ici. Cependant vous pouvez trouver des exemples typiques de fichiers gitignore pour différents types de projets sur Github Gitignore.
- `git rm exemple.txt` permet de supprimer un fichier de git et de votre dossier. Si vous le supprimez tout simplement de votre dossier git le considèrera comme ne faisant pas parie de l'index.

Le travail en remote avec git

Pour collaborer avec d'autres personnes sur des projtets, vous devez pouvoir gérer vos dépôts distants. Les dépôts distants sont des versions de vos projets qui sont hébergés sur un réseau ou sur inernet. Git étant un VCS distribué, il permet également de travailler avec des dépots distants. la commande `git clone url/du/dépot/distant` est la première étape pour récupérer un dépôt distant sur votre machine. La commande

`git remote` vous permet de lister les alias des dépôts distants spécifés. Lorsque vous clonez un dépôt distant, vous devriez *au moins* voir `origin`. C'est le nom abrégé par défaut que Git donne au serveur distant depuis lequel vous avez cloné le dépôt.

```
$ git clone https://github.com/compte/projet
Cloning into 'projet'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s,
   done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd projet
$ git remote
origin
```

En spécifiant le paramètre `-v` vous pouvez voir les URLs correspondant à chaque alias (nom abrégé):

```
$ git remote -v
origin https://github.com/compte/projet (fetch)
origin https://github.com/compte/projet (push)
```

Le workflow avec un dépôt distant:

Associer le dépôt local au dépôt distant

Imaginez que vous travaillez sur un projet sur votre ordinateur. Vous ne l'avez pas cloné depuis un dépôt en ligne et vous voulez l'envoyer sur un dépôt de votre compte github. D'abord il faudra spécifier à git l'URL du dépôt en question. Pour le faire on utilise la commande:

```
$ git remote add <alias> <url>
```

- L'alias est un nom abrégé qui permet de désigner le dépôt distant par un nom plus commode au lieu d'avoir à utiliser son URL à chaque fois;
- l'URL est le lien qui conduit vers le dépôt distant en question.

Donc la commande `git remote add origin https://github.com/compte/projet` par exemple ajoute à votre projet le dépôt distant nommé `origin` et dont l'URL

est `https://github.com/compte/projet`. A cette étape nous venons juste de dire à git *"Associe ce dépôt distant à mon dépôt local et donne lui le nom `origin` pour que je puisse le désigner facilement"*. Il n'y a eu aucun transfert de données.

- Il est possible d'associer votre dépôt local à plusieurs dépôts distants. La commande `git remote` les listera tous.
- le nom `origin` est juste un genre de convention. Sinon il est possible de donner l'alias que vous désirez à votre dépôt distant.
- si vous voulez ajouter un dépôt distant, vérifiez d'abord ceux qui sont associés à votre dépôt local avec `git remote -v`. Si l'alias que vous vouliez utiliser est déjà pris, il va falloir en trouver un autre pour éviter les conflits de nom.

Ne confondez pas `git remote add` et `git clone`. - `git clone` permet de copier tout un dépôt distant avec son historique de modifications et toutes les informations relatives - `git remote add` permet juste d'associer un dépôt distant à votre dépôt local que vous avez créé.

Fetch et pull depuis vos remotes

Maintenant que le dépôt distant est associé à votre dépot local il y a deux opérations importantes à connaître: - Pour récupérer les données que vous n'avez pas à votre niveau, vous faites un `git fetch`. Git télécharge les données distantes sur votre dépôt local mais ne touche pas à votre travail ou à ce que vous êtes en train de faire. Il ne les fusionne pas avec votre travail local. - pour récupérer et fusionner automatiquement les données du dépôt distant avec votre dépot local il faut faire un `git pull`.

push vers les remote

Quand vous voulez envoyez l'historique de votre travail en local vers le dépôt distant, vous devez faire un `git -u push <remote> <branche>`. Par exemple pour faire un push sur la branche main du dépôt distant nommé origin, il faut faire

```
$ git push -u origin main
```

- lorsque vous faites un push pour la première fois, si vous écrivez simplement `git push`, git ne sait pas vers quelle branche ni quel dépôt distant envoyer vos données. écrire `git push -u origin main` permet non seulement de lui indiquer vers où vous faites le push, et en plus le paramètre `-u` permet de lui dire de garder ces informations. Ainsi les prochaines fois vous n'aurez qu'à écrire `git push` et il saura quoi faire.
- cette commande dans l'état marche seulement si vous avez les droits d'écriture sur le dépot distant en question et si personne d'autre n'a fait un push entre temps avant vous sur la même branch. Sinon votre push sera tout simplement rejeté. Avant de faire un push, lorsqu'on travaille à pluieurs sur une beanche, il faut d'abord faire un pull pour fusionner l'historique du dépôt distant et l'historique local avec un `git pull <remote> <branche`.

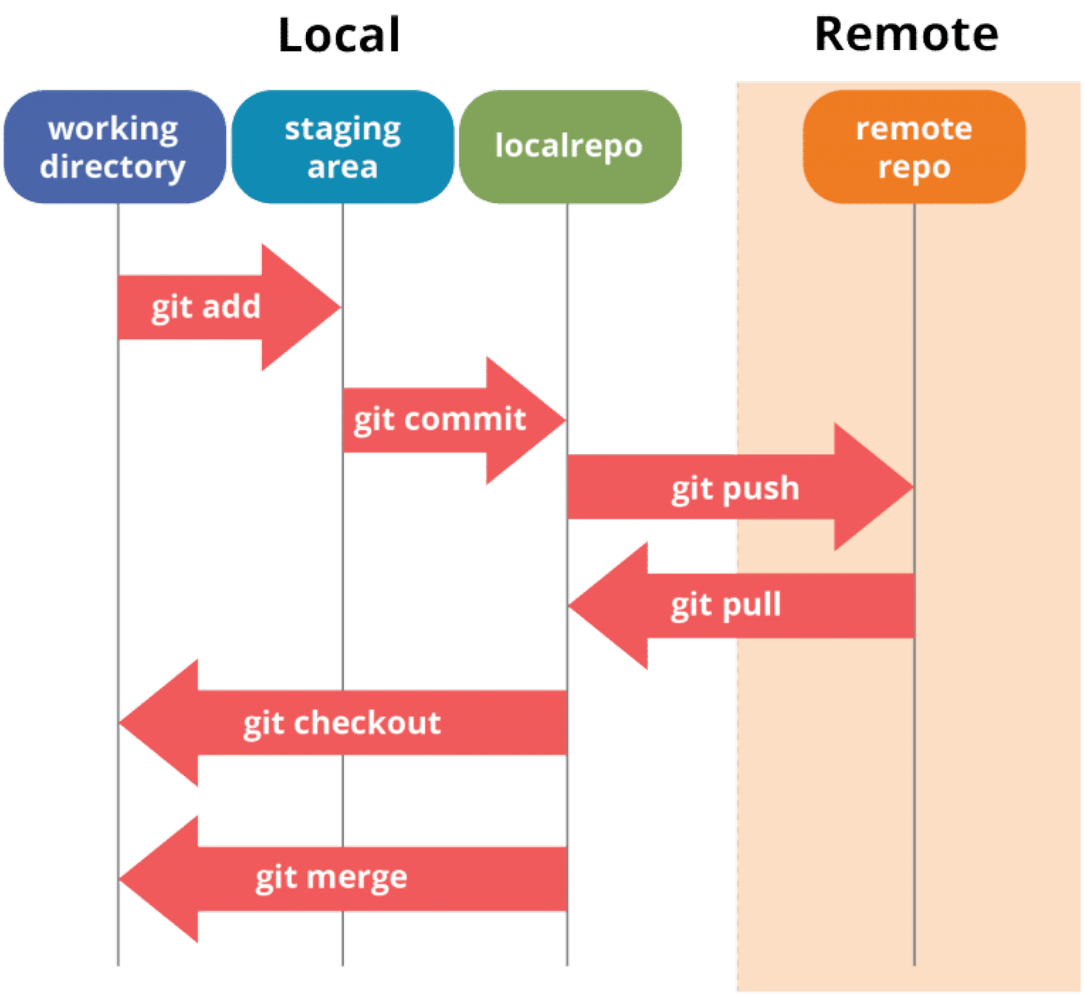

Figure 2: Les états de Git

Et… ensuite ?

Git est un outil très puissant qui possède un grand nombre de fonctionnalités. L'une de ses forces principales est la gestion des branches. Celle-ci introduit de nouveaux concepts que nous ne pourrons pas couvrir. Ceux que nous avons pu découvrir suffiront largement pour la suite de notre travail de base qui est la POO.

Que faut-il retenir ?

Nous avons vu beaucoup de choses. Si vous avez des difficultés à toutes les retenir, voici l'essentiel à connaître pour nos travaux.

Pour récupérer un dépôt

cloner le projet avec

```
$ git clone url/du/projet
```

Tout le projet avec tout son historique de versions sera copié dans le dossier correspondant.

Pour travailler sur un dépot

Si ce n'est pas un dépôt distant cloné:

1. Initialiser le contrôle de version avec git dans le dossier à surveiller avec:

```
$ git init
```

2. ajouter tous les fichiers du dossier (y compris ceux dans les dossier enfants) avec à au stage (ou index). Ils seront dans le prochain commit.

```
$ git add .
```

Notez que le point est très important (il y a un espace entre add et le point). Il dit à git "ajoute tout" pour ne pas avoir à se casser la tête.

3. faire un commit. Git sauvegardera ainsi cette version de tous vos fichiers

```
$ git commit -m "message du commit"
```

4. Ajouter le dépôt distant avec

```
$ git remote add origin url/du/depot/distant
```

5. envoyer vers le dépot distant en faisant d'abord un pull pour fusionner les historiques distants et local puis faire un push pour les envoyer.

```
$ git pull origin main
$ git push -u origin main
```

6. vous pouvez commencer votre travail. Chaque fois que vous voudrez garder votre projet tel qu'il est vous n'aurez qu'à suivre les étapes 2 à 5 (sauf la 4).

```
$ git add .
$ git commit -m "message du commit"
$ git push
```

À suivre

Ça y est, nous y EST… Enfin presque. Maintenant que vous avez les bases du versionnage avec git, nous allons enfin pouvoir entâmer la dernière ligne droite avant de commencer notre apprentissage de la POO à proprement parler. Nous allons donc étudier la prochaine fois les bases même du langage java. Les variables, les opérateurs, les conditions, les boucles les tableaux… Bref tout ce qu'il faut connaître pour bien débuter. Et nous pourrons ensuite attaquer la bête!