# Git

Source : Chat Gpt – Cours OFPPT

Collecté par : Naoual ABDALLAH

# Version Control System (VCS)

Un système de gestion de versions est un outil qui permet de gérer différentes versions d'un même projet.

Il permet de :

- D'enregistrer l'historique des changements sur un ou plusieurs fichiers (code, documents, etc.)
- Revenir à une version antérieure en cas d'erreur
- Collaborer à plusieurs sans écraser le travail des autres
- Suivre qui a changé quoi, quand, et pourquoi

Version = contenu du projet à un moment de son cycle de vie.

# Types de VCS

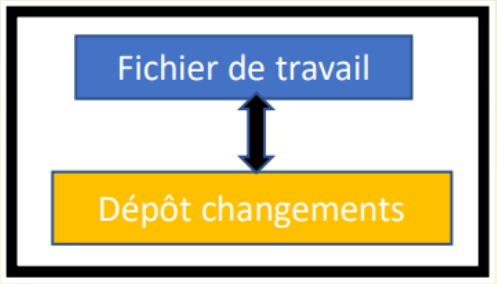
- VCS local
- VCS centralisé
- VCS distribué

# VCS local

Garde l’historique des fichiers uniquement sur la machine locale sans collaboration avec d'autres développeurs.

- Rapide car tout est local.
- Utile pour des projets personnels ou expérimentaux.
- Pas de collaboration possible.
- Pas de sauvegarde externe : si le disque dur plante, tout est perdu.

Observez le cadre noire = machine client




**RCS** (Revision Control System)

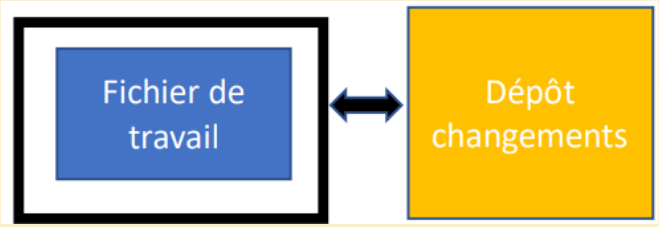
**SCCS** (Source Code Control System)

**Simple copie manuelle…**

# VCS centralisé

Un serveur central contient la version principale du projet. Les utilisateurs se connectent au serveur pour envoyer, mettre à jour ou récupérer les fichiers.

- Idéal pour les équipes avec contrôle centralisé fort.
- **Dépend fortement du serveur central**. Si le serveur tombe, **tout le travail est bloqué, l'historique est perdu**
- Moins adapté au travail hors-ligne.



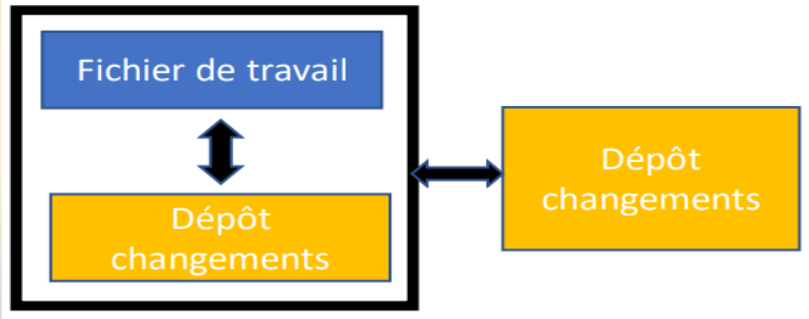

**SVN** (*Subversion*)
**CVS** (*Concurrent Versions System*)
**Perforce**
**TFS** (*Team Foundation Server*)…

# VCS distribué

Combine la gestion de version locale et centralisée en créant deux dépôts (local et distant). Chacun peut travailler à son rythme, de façon désynchronisée des autres, puis échanger les travaux avec d'autres développeurs.

- Travail hors-ligne possible.
- Meilleure gestion de la collaboration et des fusions.
- Un peu plus complexe à prendre en main.
- Peut consommer plus d'espace disque (copie complète du dépôt)

**Dépôt ou Repository ou repo**




**Git**
**Mercurial**
**Bazaar**
**Fossil**
**GNU Arch…**

# wiki

Le **principe du wiki** est que **tout le monde peut créer, modifier et améliorer le contenu collaborativement**, directement depuis un navigateur web.

**L'objectif est de créer une base de connaissances vivante.** Le but n'est pas la perfection immédiate, mais **l'amélioration continue**. Chacun peut corriger ou compléter ce que d'autres ont écrit. Le contenu s'affine alors avec le temps.

- **Collaboration ouverte :** Un wiki permet à plusieurs personnes (souvent une communauté) de **travailler ensemble sur le même contenu ( exemple wikipedia**, où des millions d'utilisateurs enrichissent collectivement les articles).
- **Historique des modifications :** Chaque changement est **enregistré** avec la date, l'auteur et la et la version du contenu avant et après la modification (retour en arrière possible)

# wiki

- **Structure hypertexte simple :** Les wikis utilisent souvent une **syntaxe légère** (comme Markdown ou une syntaxe wiki propre) pour formater le texte et créer des **liens entre les pages** facilement
- **Ouverture et contrôle :** Même si l'esprit du wiki est collaboratif et ouvert, il existe des **niveaux de permissions** (totalement publics (ex. : Wikipedia), restreints à une équipe ou une entreprise (ex. : wikis internes comme Confluence, Notion, ou GitLab Wiki)

**Le wiki est un système centralisé de gestion collaborative de contenu**, avec un suivi des versions intégré.

# Git

Git est un VCS distribué et libre. Il permet à un utilisateur de créer différentes versions de son travail en local, d'envoyer ce travail vers un serveur central pour collaborer avec d'autres utilisateurs, mais aussi de récupérer les modifications effectuées par ces derniers.

# Git

WIKi?? p7


Exemple de gestion de versions distribuée source https://edutechwiki.unige.ch/

# Installation et configuration

- Pour connaitre la version installée :

  git -- version

- Pour l'installer :

  https://git-scm.com/install/

- Avant de pouvoir utiliser les commandes GIT vers un serveur distant, il faut fournir certaines informations de configuration :

  git config --global user.email "email@example.com"

  git config --global user.name "nom_utilisateur"

Le paramètre --global permet de conserver les informations pour tous les prochains projets.

# Commandes shell

**Commandes Unix/Linux** utilisées dans Git Bash fournit sous Windows :

- Créer un dossier : mkdir chemin
- Changer de dossier : cd Chemin
- Consulter le contenu d'un dossier : dir / dir chemin / ls / ls chemin
- Pour aller au dossier parent : cd ..
- Ajouter des données à un fichier :

  echo "votre texte" >> nom_fichier_existant.txt
- Efface le contenu du fichier et lui ajouter des données :

  echo "Votre texte" > nom_fichier.txt
- Ouvrir et éditer un fichier en utilisant un éditeur vi :

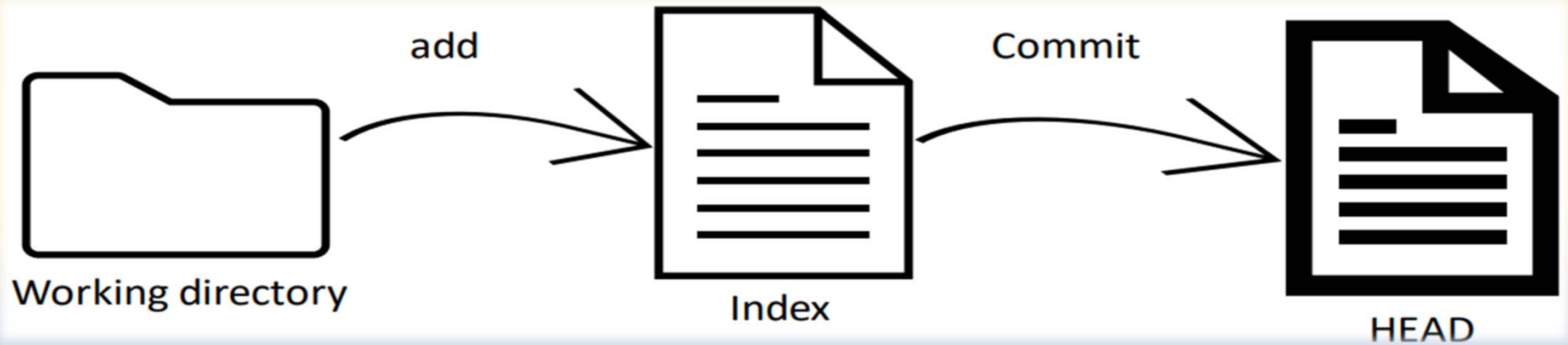
  vi nom_fichier.txt (i : insert / esc + :wq = sortir / esc + :q! Sortir sans enregistrer)

# Dépôt local

**Dépôt(repository) = l'historique du projet, contenant toutes ses versions.**

Lorsqu'on travaille avec Git, il est essentiel de faire la distinction en local entre trois zones :

- working directory (répertoire de travail ou working tree)
- Index (espace d'indexation ou staging area)
- Head



# Working directory

C’est le dossier de travail où l’utilisateur stocke ses fichiers de projet sur son disque dur. Il peut les voir, les modifier, les supprimer et même en créer de nouveaux.

Pour activer un système de gestion de version sur son espace de travail en local, l’utilisateur doit exécuter l’instruction :

**git init**

Un dossier caché .git est créé. l’espace de travail est maintenant devenu un dépôt local. Il a la capacité de stocker plusieurs versions du projet en cours.

# Index

La zone d’indexation est une zone intermédiaire (espace de transit) où seront placées les modifications à apporter lors du prochain commit.

Un **commit** en Git est une capture instantanée de l’état du projet à un moment donné. Il enregistre les changements (ajouts, suppressions ou modifications de fichiers) dans l’historique de Git.

# Index

**Pour faire entrer des éléments dans l'espace d'indexation :**

- **Fichiers cités :**

  **- git add fichier1 fichier2…**

- **Fichiers et dossiers dans la racine du répertoire courant (sauf ceux dont le nom commence par un .)**

  **- git add ***

- **Fichiers du répertoire courant et tous ses sous-répertoires**

  **- git add .**

**Pour faire sortir le fichier de la zone Index**

**- git rm -- cached nom_fichier  ou git restore --staged nom_fichier**

# Index

**Pour faire un commit :**

**- git commit –m "message explicatif"**

**Pour faire un commit à tout ce qui se trouve dans le dossier (sans utiliser add) :**

**- git commit -a -m "message"  (le a pour all)**

Mais **cela n'inclut pas les nouveaux fichiers non suivis**.

# Vérification de l’état

- **Pour afficher l’historique des commits :**

  **- git log**

- **Pour afficher l’historique des modifications d’un fichier précis, même s’il a été renommé ou déplacé au cours du temps :**

  **- git log --follow nomfichier**

- **Pour afficher les différences entre le répertoire de travail, l'index et HEAD :**

  **- git status**

  ( qu’est ce qui n’a jamais été ajouté dans l’index, ce qui n’est pas dans l’index, ce qui a été modifié et non encore commité, … )

# Vérification de l'état

- **Pour remettre un fichier modifié dans son état d'origine** (celui du dernier commit) **:**

  **- git restore <nomdufichier>**

  **- git checkout -- <nomdufichier>**

  (les changements sont non récupérables. Les changements déjà ajoutés à l'index seront gardés.)

- **Pour retirer un fichier de la zone d'indexation :**

  **- git restore --staged <nomdufichier> pour le retirer de l'index**

- **Pour récupérer une version spécifique en local**

  **- git checkout code_version nom-fichier**

Code version à récupérer avec git log

# Head

- C’est un **pointeur symbolique** vers le commit courant.
- Indique sur quelle version du projet, on est actuellement positionné.
- HEAD change, suite à chaque nouveau commit.
- HEAD change, à chaque passage de version en version

# Branches

Une branche (*branch*) est une variante du projet (une version parallèle), utilisée pour développer des fonctionnalités, corriger des bugs ou expérimenter, sans impacter la branche principale (souvent appelée main ou master).

Une branche est un pointeur mobile vers un commit spécifique dans l'historique du projet. Elle permet de :

- Travailler en parallèle sur plusieurs évolutions du code.
- Garder la branche principale stable.
- Fusionner les changements plus tard si souhaité (via merge ou rebase).

# Manipulation des branches

- Afficher la liste des branches :
  - **git branch**
- Créer une branche sans changer d'emplacement :
  - **git branch nom_nouvelle_branche**
- Aller à une branche :
  - **git checkout nom_branche ou git switch nom_branche;**
- Créer une branche et se positionner sur cette nouvelle branche :
  - **git checkout -b nouvelle_branche ou git switch -c nouvelle-branche**
- Supprimer une branche :
  - **git branch -d nom_branche**
- Renommer une branche :
  - **git branch -M nouveau_nom**
- Fusionner une branche dans la branche actuelle :
  - **git merge nom_branche**

**Une branche n'est pas disponible pour les autres tant que vous ne l'aurez pas envoyée vers votre dépôt distant**

# Vérification de l’état

La commande **git diff** permet de **voir les différences** entre deux versions du code. Elle  montre **ce qui a été ajouté, supprimé ou modifié**.

Pour voir Les modifications **non ajoutées (Working directory** vs **Index)**

**- git diff**

**Pour voir les modifications ajoutées avec git add, mais pas encore commités (Index** vs **HEAD)**

**- git diff --staged**

**Pour voir les différences entre ton répertoire actuel et le dernier commit (Working directory** vs **HEAD)**

**- git diff HEAD**

**Pour voir la différence entre deux commits précis :**

**- git diff version1 version2**

Les lignes commençant par : + ➜ ajoutées, - ➜ supprimées

# GitLab vs GitHub

**GitLab et GitHub** sont des exemples de plateformes **open source** et **collaboratives** de développement basée sur Git. Elles permettent d'héberger des projets web, du code source, ainsi que de la documentation.

# GitLab vs GitHub

| Critère | GitLab | GitHub |
|---|---|---|
| Propriétaire | **GitLab inc.** | Microsoft |
| Utilisation | Hébergement de code open source et projets collaboratifs | Gestion complète du cycle de développement (code, tests, déploiement) |
| Tarification | Gratuit pour beaucoup de fonctions, même en interne | Gratuit pour projets publics, payant pour certaines fonctions privées |
| Popularité | Moins populaire, mais en forte croissance | Le plus utilisé au monde |
| Communauté | Moins grande, mais active dans le monde open source | Très grande communauté mondiale, énormément de projets open source |
| Public idéal | Équipes d'entreprise, projets internes | Communauté open source, développeurs individuels |

# GitLab vs GitHub

| Critère | GitLab | GitHub |
|---|---|---|
| Interface | Excellente intégration (VS Code, Azure, Slack, etc.) | Plus simple, plus conviviale pour les développeurs |
| Fonctionnalités intégrées | Outil tout-en-un : planification, CI/CD, sécurité, déploiement | Centré sur le code et la collaboration |
| Confidentialité | Très bon pour projets privés et internes | Très bon pour projets publics |
| CI/CD (automatisation de tests et déploiements) | Intégré directement dans GitLab | Nécessite GitHub Actions (outil séparé) |
| Intégrations | Intègre déjà beaucoup d'outils en interne (Jira, Slack, Kubernetes, etc.) | Fonctionne très bien avec de nombreux outils externes (VS Code, Azure, Slack, etc.) |
| Gestion de projets | Outils intégrés : kanban, milestones, time tracking | Projets via **GitHub Projects** (kanban amélioré) |

# Communication avec un repository distant

Le **repository distant** est hébergé sur **GitLab ou GitHub** .

Pour associer un repository local existant à un repository distant :

**- git remote add nom_local_dépôt  adresse_dépôt_distant**

Exemple :

**- git remote add origin https://gitlab.com/utilisateur/mon-projet.git**

**origin** : C'est le nom donné en local au repository distant

- Le dépôt principal est appelé origin (c'est une convention).
- Il est possible de fournir un autre nom

(ex. git remote add gitlab https://gitlab.com/...)

- **L'adresse du repository distant** est fournie par GitLab lors de la création du projet.
- Pour renommer le nom : **git remote rename old_name new_name**

# Communication avec un repository distant

git clone sert à **copier un dépôt Git distant existant** sur l'ordinateur local.

Elle télécharge **tous les fichiers**, **l'historique des commits**, et **les branches** du projet.

**- git clone adresse_dépôt_distant**

Elle crée automatiquement un dépôt local sur la machine locale portant le nom du projet (déjà initialisé)

# Envoi des données vers le repository distant

- Pour publier des modifications de la branche actuelle vers la branche distante :

  **- git push**

- Pour publier des modifications depuis une branche vers le dépôt distant

  **- git push nom_local_dépôt nom_branche**

- Pour publier toutes les branches locales vers le dépôt distant

  **- git push --all origin**

- Pour supprimer une branche sur le dépôt distant

  **- git push nom_dépôt_distant --delete nom_branche**

- Force le push (dangereux si travail collaboratif) même si l'historique des versions sur le dépôt distant est différent de celui sur le dépôt local

  **- git push --force**

# Récupération des données depuis le repository distant

- pour récupérer les changements distants sans les fusionner vers head :

  **- git fetch**

  - Se connecte au dépôt distant
  - Télécharge les nouveaux éléments manquants
  - Met à jour les branches distantes locales
  - **Ne change pas la branche locale courante** (comme main)

- pour récupérer les changements distants et les fusionner vers head :

  **- git pull**

  - Se connecte au dépôt distant
  - Télécharge les nouveaux éléments manquants
  - Met à jour les branches distantes locales
  - **Fusionne ces changements avec la branche locale courante** (comme main)

# Merge Request

Merge Request (Les demandes de fusion) : sont la façon dont vous vérifiez les modifications du code source dans une branche. Lorsque vous ouvrez une demande de fusion, vous pouvez visualiser et collaborer sur les changements de code avant la fusion.

- Vous pouvez créer une Merge Request à partir de la liste des demandes de fusion.

1. Dans la barre supérieure, sélectionnez Menu > Projets et recherchez votre projet:
2. Dans le menu de gauche, sélectionnez Merge Requests.
3. puis, sélectionnez new merge request.
4. Sélectionnez une branche source et cible, puis Comparez les branches et continuez.
5. Remplissez les champs et sélectionnez create merge request

# Merge Request

Il est possible de créer un Merge Request en exécutant des commandes Git sur votre ordinateur local.

1.Créer une branche: git checkout -b my-new-branch

2.Créer ou modifier vos fichiers ,puis les mettre dans le stage et les valider :

```
git add .
git commit -m " message de valdation"
```

3.Poussez votre branche vers GitLab : git push origin my-new-branch

4.GitLab vous invite avec un lien direct pour créer une demande de fusion : Copiez le lien et collez-le dans votre navigateur.

# Communication avec un repository distant

Un **dépôt Git local** peut être **lié à plusieurs dépôts distants**. C'est très pratique pour **travailler avec plusieurs plateformes** (ex. GitHub, GitLab, Bitbucket) ou pour avoir une **copie de sauvegarde** du code.

- **git remote :** Liste les noms des dépôts distants
- **git remote -v** : Liste les dépôts distants **avec leurs URL**

Exemple affichage :

```
origin  https://github.com/utilisateur/projet.git (fetch)
origin  git@github.com:utilisateur/projet.git (push)
```

Dans Git, l'URL pour fetch peut être différente de l'URL pour push. On peut par exemple utiliser https pour fetch et ssh pour push:

```
git remote set-url origin https://github.com/utilisateur/projet.git  # fetch HTTPS
git remote set-url --push origin git@github.com:utilisateur/projet.git    # push SSH
```

# Annuler les modifications

- Si à la place vous voulez supprimer tous les changements et validations locaux, récupérez le dernier historique depuis le serveur et pointez la branche principale locale dessus comme ceci :
  - **git fetch origin**
  - **git reset --hard origin/master**

Après git fetch origin + git reset --hard origin/master : ton dépôt local **ressemble exactement au dépôt distant**, comme si tu venais de cloner.

# Fork

En **GitLab**, *forker un projet* (ou faire un **fork**) veut dire **créer une copie d’un projet existant** dans ton propre espace.

Ceci est utile lorsque vous souhaitez contribuer au projet de quelqu'un d'autre ou démarrer votre propre projet basé sur le sien.

fork n'est pas une commande dans Git, mais quelque chose d'offert dans GitLab et d'autres référence de version, pour pouvoir y travailler **sans modifier l’original**.

# Projet gitlab vs Fork gitLab

| Terme | Définition | Qui le possède ? | Objectif principal | Peut modifier l'original ? |
|---|---|---|---|---|
| **Projet** | Le dépôt (repository) **d'origine** sur GitLab, où le code est stocké et développé. | Le **créateur** ou l'**équipe d'origine** | Développer et maintenir le code "officiel" | ✅ Oui |
| **Fork** | Une **copie du projet** dans ton propre espace GitLab. | **Toi** (ou ton groupe) | Tester, modifier ou proposer des améliorations sans toucher à l'original | ❌ Non, sauf si tu proposes une merge request |

# Fork

Par défaut seul le créateur d'un dépôt peut y ajouter des commits.

Si un autre développeur souhaite contribuer au dépôt, il existe deux manières de procéder:

- forker le dépôt puis proposer une contribution à l'aide d'un pull request(merge request);

- Obtenir la permission d'ajouter des commits directement dans le dépôt.

**(Il faut que le projet autorise les push)**

# Rôle des membres d'un projet

| Rôle | Ce qu'il peut faire |
|---|---|
| **Guest** | Voir le projet, créer des issues, commenter |
| **Reporter** | Lire le code, télécharger les fichiers, créer des issues |
| **Developer** | **Faire des commits et push**, créer des branches, créer des merge requests |
| **Maintainer** | Tout ce qu'un Developer peut faire + gérer les paramètres du projet |
| **Owner** | Tous les droits sur le projet |

**Project / manage members**

# Fork et merge

- Tu trouves un projet intéressant sur GitLab.
- Tu **forkes** le projet → il apparaît dans ton espace GitLab (comme une copie).
- Tu **clones** ton fork → tu l'installes sur ton ordinateur pour modifier le code.
- Tu fais des changements localement, puis tu les envoies sur ton fork.
- Tu peux ensuite proposer ces changements au projet original via une **merge request**.
- 👉 **En résumé :**
- **Fork = copier sur GitLab.**
- **Clone = copier sur ton ordinateur.**

# Conflits de fusion

- Avec git pull et git merge, git tente d'auto-fusionner les changements. Malheureusement, ça n'est pas toujours possible et résulte par des conflits.
- Un **conflit** se produit quand Git n’arrive pas à fusionner automatiquement deux branches, parce que **le même fichier** (ou la même ligne) a été **modifié différemment** dans chaque branche. Git te demande alors de **résoudre manuellement** le conflit.

# Conflits de fusion

| Type de conflit | Cause | Comment résoudre |
| --- | --- | --- |
| **Conflit sur le contenu d'un fichier** | Deux branches ont modifié **la même ligne** | Ouvrir le fichier, repérer les marqueurs <<<<<<<, =======, >>>>>>>, choisir ou combiner les modifications, puis : git add <fichier> et git commit |
| **Conflit d'insertion (ajouts au même endroit)** | Deux ajouts différents au même endroit dans un fichier | Comme ci-dessus : décider de l'ordre ou combiner les ajouts, git add <fichier> et git commit |
| **Fichier renommé / déplacé** | Une branche a renommé ou déplacé un fichier et l'autre l'a modifié | Vérifier le nom/fichier final souhaité, renommer ou déplacer si nécessaire, git add <fichier> puis git commit |

# Conflits de fusion

| Type de conflit | Cause | Comment résoudre |
| --- | --- | --- |
| **Conflit de suppression** | Un fichier est supprimé dans une branche mais modifié dans l'autre | Choisir si le fichier doit rester ou être supprimé, puis git add <fichier> (si conservé) ou git rm <fichier> (si supprimé), puis git commit |
| **Conflit de mode de fichier** | Permissions ou type de fichier modifié sur deux branches | Corriger les permissions ou type de fichier selon ce qui est voulu, puis git add <fichier> et git commit |
| **Conflit sur submodule** | Deux branches ont modifié la référence du submodule différemment | Aller dans le dossier du submodule, choisir la version du commit voulu (git checkout <commit>), puis revenir au dépôt principal, git add <submodule> et git commit |

# Scénario de conflit

1. Créer un nouveau dépôt sur GitLab
2. Cloner ce dépôt localement avec git clone
3. Dans le répertoire du dépôt local, créer un fichier test.txt contenant un texte
4. Créer un commit initial sur la banche master et l'envoyer sur GitLab (git add puis git commit puis git push)
5. Créer une branche branche1 à partir de master, avec git checkout -b
6. Dans cette branche, modifier le premier texte dans test.txt, créer un commit, puis envoyer les modifications de cette branche sur GitLab
7. Revenir à la branche master avec git checkout
8. Créer une branche branche2 à partir de master (comme dans l'étape 5)
9. Dans cette branche, modifier le texte du fichier test.txt différemment de celui saisi à l'étape 6, créer un commit, puis envoyer les modifications de cette branche sur GitLab
10. Revenir à la branche master
11. Fusionner branche1 dans master, avec git merge
12. Fusionner branche2 dans master
13. Taper git status pour voir le message du conflit Etap

# Scénario de conflit

**Avant tout merge :**

- master contient la version originale de test.txt.
- branche1 contient une version modifiée de test.txt.
- branche2 contient une autre version modifiée de test.txt (différente de branche1).

**Quand tu merges branche1 dans master :**

- Aucun conflit, car master n'avait pas encore de modification concurrente sur test.txt.
- master est maintenant identique à branche1.

**Quand tu merges branche2 dans master ensuite :**

- Problème ❗ branche2 a aussi modifié le même fichier test.txt, **au même endroit**, **mais différemment**.
- Git essaie de fusionner automatiquement, mais il ne sait pas **quelle version choisir** :
  - Celle de master (issue de branche1)
  - Ou celle de branche2.

- Il y a conflit parce que les deux branches ont modifié le même fichier, sur les mêmes lignes, de manière différente.
  Git ne peut pas deviner quelle version tu veux garder, donc il te demande d'intervenir manuellement.

# Scénario de conflit

Pour résoudre ce conflit, il va falloir:

1. ouvrir le fichier test.txt dans son éditeur de code,

2. constater comment git représente le conflit, et la source de chaque version,

3. éditer le fichier pour ne conserver que la version finale souhaitée, 4. puis créer un commit.

5. Après la résolution du conflit de fusion, taper git status pour voir le résultat

6. supprimer les branches branche1 et branche2, dans votre dépôt local, et dans le dépôt distant associé (sur GitLab)

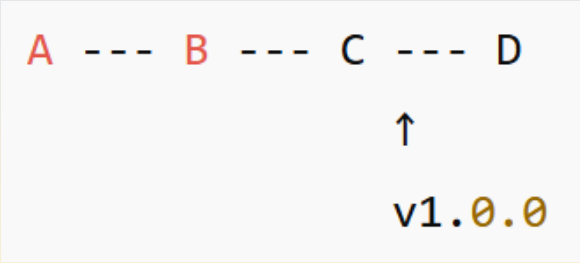
# Commit vs Tag

- Un **commit**, c'est une **sauvegarde du code** à un moment donné, avec :
- un identifiant unique (SHA, comme a7b3f2c)
- un message ("ajout de la page de connexion")
- les modifications exactes effectuées
- Chaque commit forme une **chaîne chronologique** :
    - commit A : init du projet
    - commit B : ajout du module de connexion
    - commit C : correction du bug de login
    - commit D : ajout du paiement


un commit = un petit pas dans l'évolution du projet.

# Commit vs Tag

- **Le tag : un “repère important”**
- Un **tag** ne crée rien de nouveau dans le code. Il **pointe simplement vers un commit précis** pour le **nommer** ou le **repérer facilement**.
- git tag -a v1.0.0 -m "Première version stable«




Ce tag dit :“C’est la version 1.0.0 officielle du produit.”

- git push origin v1.0.0
- git checkout v1.0.0