# Contrôle de version avec GIT et GitHub

**Fahchouch Mohammed**

2024-2025

# Historique de Git

Git est un système de gestion de versions développé en 2005 par **Linus Torvalds**, le créateur du noyau Linux. Son développement a été motivé par la nécessité de remplacer un logiciel propriétaire appelé **BitKeeper**, qui était utilisé par la communauté du noyau Linux à l'époque.

Lorsque BitKeeper a retiré sa licence gratuite à la communauté open source, cela a créé une urgence : il fallait un outil de gestion de versions qui soit **rapide**, **fiable**, **distribué**, et surtout **open source**. En réponse à ce besoin, Linus Torvalds a conçu Git avec les priorités suivantes :

- Une **excellente performance**, même pour les très grands projets.
- Un système **entièrement distribué**, où chaque contributeur possède une copie complète de l'historique du projet.
- Une **intégrité forte des données**, assurée par le stockage des données avec un système de hachage cryptographique (SHA-1).
- Une grande souplesse pour prendre en charge différents flux de travail.

Rapidement adopté par la communauté open source, Git est aujourd'hui l'outil de gestion de versions le plus utilisé dans le monde, notamment grâce à des plateformes comme GitHub, GitLab et Bitbucket. Il est utilisé aussi bien dans les petites équipes de développement que dans les grandes entreprises technologiques telles que Google, Microsoft, Amazon ou Meta.

# Définition de Git

Git est un **système de gestion de versions décentralisé** (ou *distributed version control system*, DVCS). Il permet à plusieurs personnes de **travailler simultanément sur un même projet**, tout en assurant un **suivi précis de chaque modification** effectuée sur les fichiers.

## Objectifs principaux :

- **Conserver un historique complet** des modifications apportées à un projet.
- **Faciliter la collaboration**, même à distance, entre plusieurs développeurs.
- **Permettre de revenir à des versions antérieures** en cas d'erreur ou de besoin.
- **Gérer plusieurs versions simultanées** du même projet à travers des branches.

## Fonctionnement général :

Git fonctionne en créant des **snapshots** (instantanés) de l'état d'un projet à chaque *commit*. Contrairement à d'autres outils qui enregistrent uniquement les différences entre fichiers, Git capture l'ensemble des fichiers à chaque changement, ce qui le rend particulièrement rapide et fiable.

Chaque collaborateur possède une copie complète du dépôt (le projet avec tout son historique), ce qui permet de travailler hors ligne et de fusionner les modifications plus tard, grâce à des outils puissants de gestion de branches et de fusions.

# Avantages du Git

| Fonctionnalité | Description |
| --- | --- |
| **Historique des modifications** | Permet de consulter et suivre toutes les modifications apportées au projet, avec auteur et date. |
| **Sauvegarde complète du projet** | Chaque copie locale contient l'intégralité du dépôt, assurant une redondance naturelle. |
| **Transparence du développement** | Tous les changements sont enregistrés de façon claire, ce qui favorise une meilleure traçabilité. |
| **Reproductibilité des travaux** | Il est possible de reproduire exactement une version antérieure du projet à un moment donné. |
| **Récupération rapide des erreurs** | Possibilité de revenir à une version stable après une erreur ou un bug critique. |
| **Collaboration simplifiée** | Plusieurs personnes peuvent travailler sur le même projet sans écraser le travail des autres. |

# Installation de Git sous Linux

## Debian, Ubuntu et dérivés

```
sudo apt update
sudo apt install git
```

Met à jour les paquets et installe Git depuis les dépôts officiels.

---

## Fedora

```
sudo dnf install git
```

Git est inclus dans les dépôts par défaut de Fedora. Aucune configuration supplémentaire n'est nécessaire.

---

## Arch Linux / Manjaro

```
sudo pacman -S git
```

Installe Git rapidement avec le gestionnaire pacman.

---

## Vérification de l'installation

Pour vérifier que Git a bien été installé :

```
git --version
```

Cette commande affiche la version actuelle de Git installée sur votre système.

# Configuration initiale de Git

Une fois Git installé, il est essentiel de le configurer correctement afin de pouvoir suivre les auteurs des modifications dans les projets.

## Accéder à la documentation

Git dispose d'une documentation intégrée accessible depuis le terminal :

```
man <commande>
```

- Affiche le manuel d'une commande (ex. man git). Cependant, cette documentation peut parfois être trop technique.

```
man <commande>
```

- Affiche une aide plus concise et généralement plus pratique.

## Accéder à la documentation

Avant de commencer à utiliser Git, il est nécessaire de définir le nom d'utilisateur et l'adresse e-mail. Ces informations sont utilisées pour annoter chaque commit.

```
git config --global user.name "Nom Prénom"
git config --global user.email "adresse@email.com"
```

La commande --global permet d'appliquer la configuration à l'ensemble des projets de l'utilisateur

# Concepts fondamentaux de Git

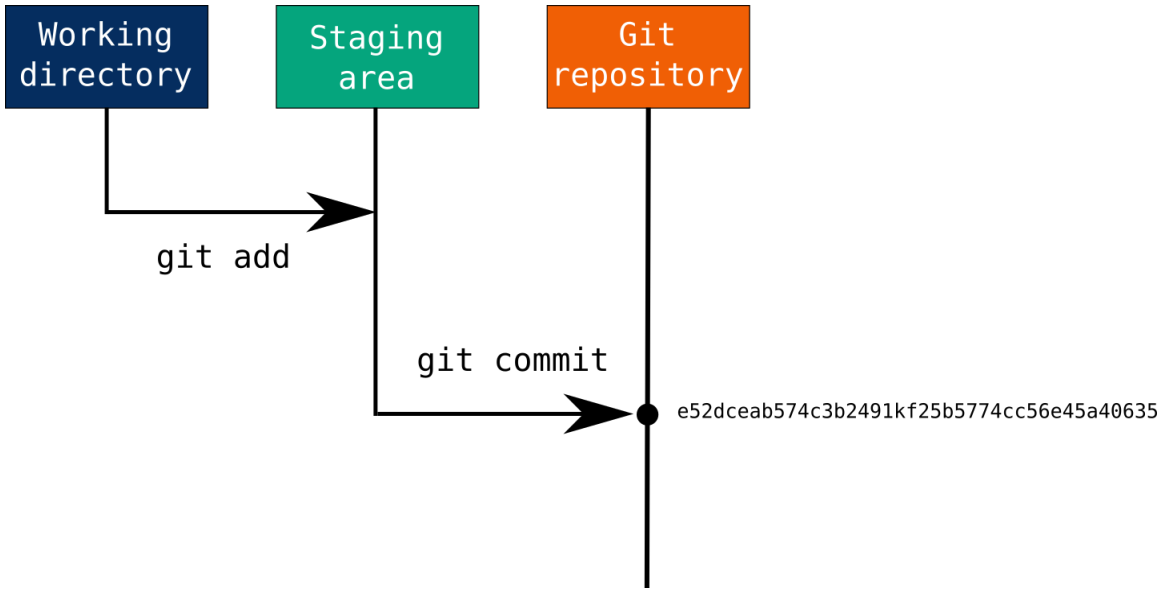
Git repose sur deux concepts clés :

## Commit

Un **commit** est une **capture des modifications** apportées aux fichiers du projet. Chaque commit enregistre un instantané des changements, avec un identifiant unique, l'auteur, et un message descriptif.

## Repository

Le **repository** est l'**historique complet** du projet, incluant tous les commits. Il peut être local (sur la machine de l'utilisateur) ou distant (par exemple, sur GitHub).



Bien sûr ! Voici la version modifiée sans numérotation dans les titres :

---

# Le cycle de vie des fichiers dans un dépôt Git

Les fichiers passent par plusieurs étapes dans Git :

### Untracked

Les fichiers non suivis par Git, présents dans le répertoire de travail.

### Modified

Les fichiers modifiés, mais non encore ajoutés à la zone de staging.

### Staged

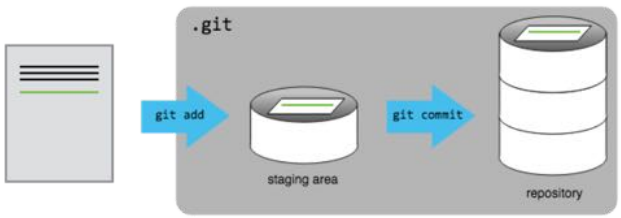
Les fichiers prêts à être enregistrés dans le prochain commit, après avoir été ajoutés à la zone de staging avec

```
git add .
```

### Committed

Les fichiers enregistrés définitivement dans le repository local après un commit avec

```
git commit .
```



# Ce qu’il faut ajouter et le fichier .gitignore

## Ce qu’il faut ajouter

Dans Git, vous devez ajouter les fichiers que vous souhaitez suivre à l’aide de la commande `git add`. Cela place les fichiers dans la **zone de staging**, prêtes à être validées dans le prochain commit.

## Le fichier .gitignore

Le fichier `.gitignore` permet d'indiquer à Git quels fichiers ou répertoires ne doivent pas être suivis. Il est souvent utilisé pour exclure :

- Les fichiers temporaires générés par l'IDE
- Les fichiers de configuration locaux
- Les dépendances ou bibliothèques externes

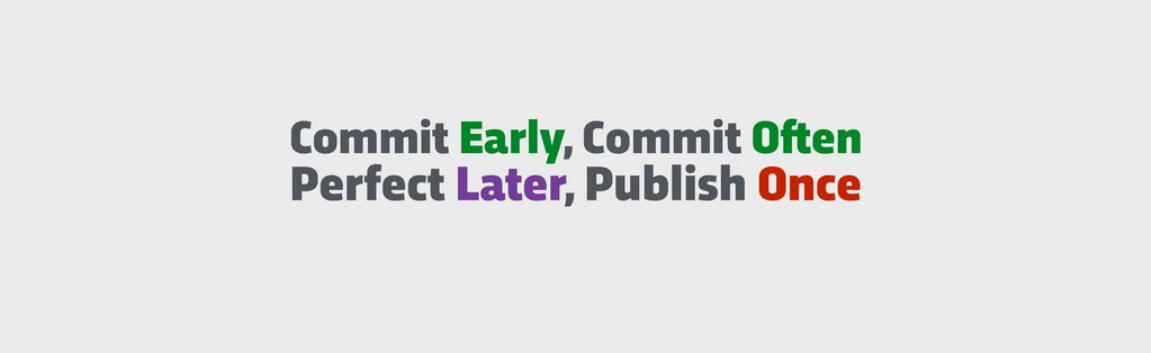
Par exemple, pour ignorer tous les fichiers `.log` et le dossier `node_modules`, ajoutez les lignes suivantes dans le fichier `.gitignore` :

```
*.log
node_modules/
```

Le fichier `.gitignore` doit être ajouté au repository afin de s’assurer que les règles d'ignorance sont partagées avec les autre contributeur.

## Quand commettre

Il est sage de commettre fréquemment, tout comme de sauvegarder régulièrement un document.
Des commits fréquents offrent plus de points de récupération.
Les bons commits sont **atomiques**, c'est-à-dire qu'ils représentent la plus petite modification significative.




## Explorer l'historique avec git log

La commande `git log` permet d'afficher l'historique des commits d'un dépôt Git, trié par ordre chronologique inverse (le commit le plus récent en premier). Elle est essentielle pour naviguer dans l'historique des modifications et voir ce qui a été changé au fil du temps.

### Principaux flags pour git log

- `-p` : Affiche les changements (diff) entre chaque commit.
- `-3` : Affiche les **3 derniers commits**. Vous pouvez remplacer `3` par n'importe quel nombre pour afficher un nombre spécifique de commits.
- `--stat` : Affiche un résumé des changements, y compris le nombre d'inserts (ajouts) et de suppressions (deletes) entre les commits.
- `--oneline` : Affiche chaque commit sur une seule ligne, montrant l'identifiant SHA-1 et le message de commit.

- **`--graph`** : Affiche un graphique simplifié de l'historique des branches, rendant la structure des commits plus visuelle.
- **`--pretty`** : Modifie le format de sortie des commits. Options possibles :
    - `short` : Affiche une sortie concise.
    - `full` : Affiche plus de détails.
    - `fuller` : Affiche encore plus d'informations détaillées.
    - `oneline` : Affiche chaque commit sur une ligne, comme `--oneline`.

#### Filtres temporels et auteurs

- **`--since=X`** : Affiche les commits faits depuis un moment spécifique. Vous pouvez préciser le délai en minutes, heures, jours, semaines, mois, ou années (par exemple, `--since=2.weeks`), ou utiliser un format de date spécifique (par exemple, `--since="2023-04-01"`).
- **`--until=X`** : Affiche les commits faits avant une certaine période. Comme `--since`, vous pouvez préciser une durée ou une date.
- **`--author=<pattern>`** : Affiche les commits réalisés par un auteur spécifique, en utilisant un motif de recherche (par exemple, `--author="John"` pour filtrer par auteur).

### Exemples pratiques

Pour afficher les derniers 5 commits, avec un graphique simplifié et les messages :

```
git log -5 --oneline --graph
```

Pour afficher les commits effectués dans les deux dernières semaines, avec les différences entre chaque commit :

```
git log --since=2.weeks -p
```

# Annuler des changements dans Git

## Annuler les modifications d'un fichier

Pour annuler les modifications non commises dans un fichier, utilisez :

git checkout -- README.md

Cela restaure le fichier à son état avant la modification. Vérifiez avec `git status` pour confirmer.

## Restaurer un fichier d'un commit spécifique

Si vous voulez revenir à une version précise d'un fichier, utilisez son hash de commit :

```
git checkout <commit-hash> README.md
```

## Modifier le message du dernier commit

Pour changer le message du dernier commit, utilisez :

```
git commit --git rm READYOU.md -m "Nouveau message"
```

## Supprimer un fichier du repository

Pour supprimer un fichier du dépôt, utilisez :

```
git rm READYOU.md
```

### Supprimer un fichier sans `git rm`

Si vous avez supprimé un fichier sans `git rm`, vous pouvez le restaurer avec :

```
git checkout -- README.md
```

### Rollback : Annuler plusieurs commits

Pour revenir à un état antérieur du projet, utilisez `git revert` pour annuler un commit spécifique, tout en conservant l'historique :

```
git revert <commit-hash>
```

# Branches dans Git

Les branches sont des versions parallèles d'un dépôt, permettant de travailler sur des fonctionnalités, des corrections de bugs ou des expérimentations sans affecter la branche principale (master). Cela permet de développer de manière isolée et de tester des idées sans perturber le travail sur la branche principale.

- **Cas d'utilisation :**
    - Développement de nouvelles fonctionnalités
    - Correction de bugs
    - Essai d'idées

Si les changements sur la branche sont réussis, vous pouvez les fusionner avec la branche principale. Si ce n'est pas le cas, il suffit de supprimer la branche.

# Flux de travail typique des branches

### Lister les branches :

```
git branch
```

Cela liste toutes les branches et montre la branche active avec un astérisque (*).

### Créer une nouvelle branche :

```
git branch test_feature
```

Cela crée une nouvelle branche appelée `test_feature`.

### Passer à la nouvelle branche :

```
git checkout test_feature
```

Vous passez alors à la branche `test_feature`.
**Modifiez, ajoutez et commitez les changements** sur la branche `test_feature`.

### Revenir à la branche principale (master) :

```
git checkout master
```

### Fusionner les changements :

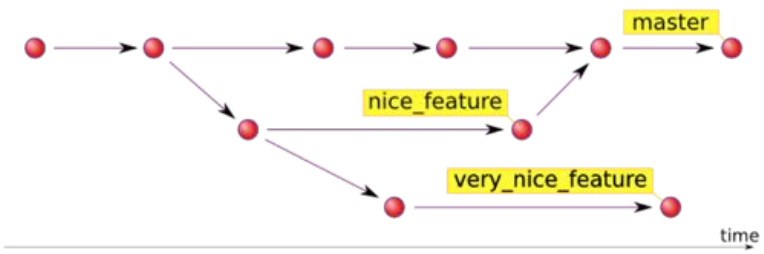
```
git merge test_feature
```

### Supprimer la branche de fonctionnalité :

```
git branch -d test_feature
```

## Collaboration avec plusieurs auteurs

- Plusieurs personnes peuvent travailler sur leurs propres branches, même si elles modifient le même fichier.
- **Les conflits** se produisent lorsque deux personnes modifient la même partie d'un fichier. Git met en évidence ces conflits et exige une résolution manuelle avant de pouvoir fusionner.



## Résolution des conflits

Si deux utilisateurs modifient la même partie d'un fichier, Git identifie cela comme un conflit :

**Exemple de conflit de fusion :**

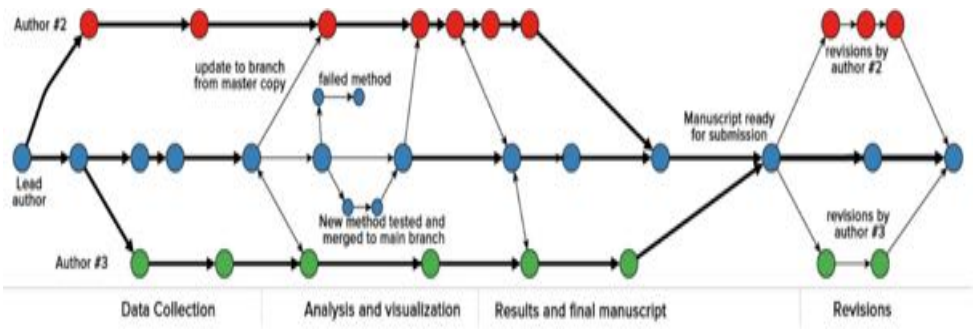
```
 CONFLICT (content): Merge conflict in README.md
La fusion automatique a échoué ; corrigez les conflits puis
validez le résultat
```

- Git marque les zones conflictuelles comme ceci :

```
 <<<<<<< HEAD
Vos modifications
=======
Modifications conflictuelles
>>>>>>> nom-de-la-branche
```

Pour résoudre le conflit :

- Ouvrez le fichier et ajustez manuellement les changements conflictuels.
- Après avoir résolu, ajoutez le fichier (`git add <fichier>`) et validez les changements.



# GitHub

GitHub est une plateforme où les développeurs stockent et gèrent leurs dépôts de code. Elle permet la collaboration, le contrôle de version et la documentation du code. On peut le considérer comme un "Facebook" pour les programmeurs — un espace pour partager, contribuer et suivre les modifications de code.

## Avantages :

- **Vrai Open Source :** Permet de collaborer sur des projets open-source.
- **Interface Graphique (GUI) :** Simplifie l'utilisation de Git pour ceux qui préfèrent une interface graphique.
- **Suivi des problèmes :** Permet de suivre les bugs et les améliorations.
- **Apprendre des autres :** Explorez le code des autres et contribuez à leurs projets.

GitHub offre également un **compte "micro" gratuit pour les étudiants pendant 2 ans**, avec un accès aux dépôts privés..

## Pourquoi utiliser GitHub ?

- **Collaboration Simplifiée :** GitHub facilite la collaboration, permettant de proposer des corrections ("Voici une correction pour ta documentation") plutôt que de simplement signaler un problème ("Il y a une faute dans ta documentation").
- **Accès aux Projets Open Source :** Voyez ce que font les autres et contribuez.
- **Interface Graphique :** Explorez le code, suivez les problèmes et naviguez dans l’historique des commits facilement

Vous pouvez connecter votre **dépôt Git local** à un ou plusieurs dépôts distants sur GitHub avec les commandes suivantes :

```
git remote add origin https://github.com/username/reponame
git remote -v  # Vérifiez vos connexions
```

```
git push origin master  # Envoyez vos modifications sur GitHub
git pull origin master  # Récupérez les modifications depuis GitHub
```

Utilisez HTTPS pour la connexion jusqu'à ce que vous soyez à l'aise avec la configuration SSH.

# Collaboration Asynchrone

### Forker un dépôt :

Forker un dépôt d'un autre utilisateur sur GitHub crée une copie sous votre compte. Cela vous permet de contribuer à leurs projets.

```
git clone https://github.com/username/reponame
```

### Faire des modifications :

Modifiez le code, ajoutez les changements et effectuez un commit dans votre dépôt local :

```
git add <fichier>
git commit -m "Description des modifications"
```

## Envoyer vos changements :

Après le commit, envoyez vos changements vers votre dépôt sur GitHub :

```
git push origin master
```

# Conclusion

GitHub est un outil essentiel pour le développement collaboratif, permettant aux développeurs de travailler sur des projets de manière asynchrone, de gérer l'historique du code et de contribuer à des projets open-source. En utilisant les fonctionnalités puissantes de GitHub telles que le forking, les Pull Requests et les branches, les développeurs peuvent collaborer efficacement, suivre les problèmes et maintenir un code propre.