

# COURS COMPLET GIT - DÉBUTANT ET AVANCÉ

---

Auteur : Paterne G. G

## 1. Introduction à Git

### Qu'est-ce que Git ?

Git est un système de gestion de versions distribué (DVCS - Distributed Version Control System). Il permet de suivre les modifications apportées à un projet informatique au cours du temps. Grâce à Git, vous pouvez revenir facilement à des versions précédentes de votre code et collaborer efficacement avec d'autres développeurs.

### Pourquoi utiliser Git ?

- Retrouver des versions antérieures du code (rollback).
- Travailler en équipe sur un même fichier de manière structurée.
- Gérer différentes branches pour expérimenter de nouvelles fonctionnalités.

---

## 2. Installation et configuration

### Installation

- **Windows / macOS** : Télécharger le programme d'installation sur [git-scm.com](https://git-scm.com).

**Linux (ex. Ubuntu/Debian) :**

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

### Configuration globale

Renseigner votre nom et votre adresse e-mail (apparaîtront dans l'historique des commits) :

```
git config --global user.name "Votre Nom"
git config --global user.email "votre.email@example.com"
```

Choisir un éditeur par défaut, par exemple (optionnel) :

```
git config --global core.editor "nano"
```

Afficher la configuration complète :

```
git config --list
```

### 3. Initialiser et gérer un dépôt

#### Initialiser un nouveau dépôt

```
git init
```

- Crée un dossier caché `.git/` contenant toutes les informations de suivi.

#### Cloner un dépôt existant

```
git clone https://github.com/utilisateur/projet.git
```

- Copie le projet (et son historique) sur votre machine.

#### Ajouter un dépôt distant

```
git remote add origin https://github.com/utilisateur/projet.git
```

- Associe votre dépôt local à un dépôt distant nommé `origin`.

#### Lister / Renommer / Supprimer un dépôt distant

```
git remote -v          # Liste les dépôts distants
git remote rename origin old # Renomme 'origin' en 'old'
git remote remove origin # Supprime le remote 'origin'
```

### 4. Suivi et validation des modifications

#### Vérifier l'état

```
git status
```

- Indique si des fichiers sont modifiés, non suivis, ou prêts à être sauvegardés (staged).

#### Ajouter des fichiers à la zone de staging

```
git add fichier.txt
git add .
```

- `git add fichier.txt` : ajoute un fichier précis.
- `git add .` : ajoute tous les fichiers modifiés et/ou nouveaux.

## Créer un commit

```
git commit -m "Description du changement"
```

- Sauvegarde de manière permanente l'état des fichiers ajoutés dans l'historique.
- **Astuce** : `git commit -am "message"` ajoute et committe d'un coup seulement pour les fichiers déjà suivis.

## Afficher l'historique

```
git log
git log --oneline
git log --graph --decorate --oneline
```

- `git log --oneline` : résumé court.
- `git log --graph --decorate --oneline` : histogramme simplifié.

# 5. Branches et navigation

## Créer et lister des branches

```
git branch          # Liste les branches locales
git branch nouvelle-branch  # Crée une nouvelle branche
```

## Changer de branche

```
git checkout nouvelle-branch
```

- Passe sur la branche `nouvelle-branch`.

## Créer et changer de branche en une seule commande

```
git checkout -b nouvelle-branch
```

## Renommer une branche

```
git branch -m ancien_nom nouveau_nom
```

## Supprimer une branche

```
git branch -d branche      # Supprime une branche fusionnée  
git branch -D branche      # Supprime une branche, même si non  
fusionnée
```

# 6. Fusion (Merge) et résolution de conflits

## Fusionner deux branches (merge)

Se rendre sur la branche **cible** (celle qui recevra les changements)

```
git checkout main
```

Fusionner la branche source :

```
git merge nouvelle-branche
```

## Conflits de merge

- Se produisent si des modifications contraires sont sur la même ligne.
- Git insère des marqueurs de conflit dans les fichiers concernés.

Il faut éditer manuellement, **choisir** quelle modification garder, puis :

```
git add fichier_conflit.txt  
git commit -m "Résolution du conflit"
```

## Merge "no fast-forward"

```
git merge --no-ff nouvelle-branche
```

- Conserve l'historique de la fusion explicitement (création d'un commit de merge).

# 7. Rebase (réécriture de l'historique)

## Pourquoi rebase ?

- Permet d'avoir un historique **linéaire**.
- Déplace les commits d'une branche au-dessus d'une autre.

## Exemple

```
git checkout ma-branche  
git rebase main
```

- "Rejoue" les commits de **ma-branche** au-dessus de **main**.

## Rebase interactif

```
git rebase -i HEAD~3
```

- Permet de modifier l'ordre, de fusionner (**squash**), renommer ou supprimer (**drop**) des commits.

## Si conflit

- Résoudre manuellement, puis :

```
git add fichier_conflit.txt  
git rebase --continue
```

- Pour annuler un rebase en cours :

```
git rebase --abort
```

**Attention** : Évitez de rebaser des branches déjà partagées avec d'autres personnes.

---

## 8. Comparaison et analyse

### Différences

```
git diff                # Comparaison des changements non validés  
git diff --staged       # Comparaison des changements "staged"  
git diff main branche  # Comparer deux branches
```

### Blame

```
git blame fichier.txt
```

- Montre, ligne par ligne, quel commit et quel auteur ont fait chaque modification.

## Bisect

```
git bisect start
git bisect bad HEAD
git bisect good <commit_ou_tag>
```

- Aide à trouver le commit qui a introduit un bug en procédant par recherche binaire.

## Log avancé

```
git log --oneline --graph --decorate
git log -p
```

git log --oneline --graph --decorate git log -p

- `-p` : montre en détail les modifications associées à chaque commit.

---

# 9. Annuler ou corriger des commits

## Revert

```
git revert <commit_hash>
```

- Crée un nouveau commit qui annule les changements du commit ciblé (l'historique est préservé).

## Reset

```
# Reculer d'un commit en gardant les changements
git reset --soft HEAD~1

# Reculer d'un commit en supprimant les changements
git reset --hard HEAD~1
```

- **Attention** : `--hard` efface définitivement les modifications.

## Amend

```
git commit --amend
```

- Permet de modifier le dernier commit (message ou fichiers inclus).

---

## 10. Stash : mettre de côté des modif

### Créer un stash

```
git stash
```

- Enregistre temporairement les modifications non commit.

### Lister les stash

```
git stash list
```

### Restaurer un stash

```
git stash pop
```

- Récupère et supprime le dernier stash.

### Appliquer un stash sans le supprimer

```
git stash apply
```

## 11. Dépôts Distants et Collaboration

### Récupérer des modifications

```
git fetch
```

- Récupère l'historique distant sans fusionner.

### Fusionner les modifications distantes

```
git pull
```

- Équivaut (par défaut) à `git fetch + git merge`.

## Envoyer ses modifications

```
git push
```

- Envoie la branche courante sur le dépôt distant.

## Suivre une branche distante

```
git push -u origin ma-branche
```

## Fork & Pull Request (via GitHub)

1. **Fork** un dépôt depuis GitHub (copie sur votre compte).
2. **Cloner** votre fork en local.
3. Créer une nouvelle branche, faire vos modifications et les pousser.
4. Sur GitHub, créer une Pull Request pour proposer vos changements au projet original.

# 12. Gestion des tags

## Créer et lister des tags

```
git tag  
git tag -a v1.0 -m "Version 1.0 stable"
```

- `-a` crée un tag annoté (avec un message).

## Pousser les tags vers le dépôt distant

```
git push origin --tags
```

## Supprimer un tag

```
git tag -d v1.0  
git push origin :refs/tags/v1.0
```

# 13. Submodules (Sous-modules)

## Ajouter un sous-module



```
git submodule add https://github.com/utilisateur/projet-sous-module.git
```

- Ajoute un autre dépôt Git à l'intérieur du dépôt actuel.

## Initialiser et mettre à jour les sous-modules

```
git submodule init  
git submodule update
```

## Cloner un projet avec ses sous-modules

```
git clone --recurse-submodules <url_du_projet>
```

# 14. Commandes avancées et utilitaires

## Cherry-pick

```
git cherry-pick <commit_hash>
```

- Copie un commit précis dans la branche courante.

## Git reflog

```
git reflog
```

- Montre un journal complet des actions (commits, checkouts), même celles qui ne sont plus dans le log normal.

## Git gc (garbage collector)

```
git gc
```

- Nettoie et optimise la base de données de Git.

## Git archive

```
git archive --format=zip HEAD > projet.zip
```

- Crée un fichier ZIP du contenu du dépôt (sans l'historique).

Git shortlog

```
git shortlog
```

- Montre un résumé des commits par auteur.

Git filter-branch (ancien, use [git filter-repo](#) si disponible)

- Permet de réécrire l'historique pour supprimer/modifier certaines informations dans tous les commits (par ex. retirer un fichier sensible).

15. Bonnes pratiques

1. **Commits fréquents** : des changements atomiques et cohérents.
2. **Messages de commit clairs** : expliquer le "pourquoi" et non seulement le "quoi".
3. **Branches descriptives** : `feature/auth`, `fix/bug-123`.
4. **Revue de code** : utilisation des Pull Requests sur GitHub pour faire relire le code par un pair.
5. **Éviter de réécrire l'historique partagé** : rebase ou reset sur des branches collaboratives peut provoquer des problèmes.
6. **Garder un `.gitignore`** : éviter d'ajouter des fichiers inutiles (ex. binaires, dépendances locales).

Exemple d'un `.gitignore` pour un projet Node.js :


```
node_modules/  
.env  
*.log
```

Voici un tableau complet de toutes les commandes Git présentées, organisé avec cinq colonnes :

1. **Commande**
2. **Utilisation** (syntaxe rapide)
3. **Explication** (à quoi elle sert)
4. **Avantage**
5. **Inconvénient**

Commande	Utilisation	Explication	Avantage	Inconvénient
<code>git init</code>	<code>git init</code>	Initialise un nouveau dépôt Git dans le dossier actuel (création du dossier <code>.git/</code> ).	Démarrer facilement un nouveau projet.	Crée un dépôt vide (peut nécessiter ensuite config/dépôt distant).



Commande	Utilisation	Explication	Avantage 	Inconvénient 
<b>git clone</b>	<code>git clone URL [dossier]</code>	Copie un dépôt distant (ou local) dans un nouveau dossier.	Récupération rapide d'un projet existant et de son historique.	Peut télécharger un grand volume de données si le repo est volumineux.
<b>git remote add</b>	<code>git remote add &lt;nom&gt; &lt;URL&gt;</code>	Associe un dépôt distant (ex: <b>origin</b> ) à l'URL spécifiée.	Permet de pousser/puller vers un dépôt en ligne (ex. GitHub).	Erreur possible si on se trompe d'URL.
<b>git remote rename</b>	<code>git remote rename &lt;old&gt; &lt;new&gt;</code>	Renomme un dépôt distant existant (ex. <b>origin</b> → <b>oldorigin</b> ).	Clarifie le nom d'un remote (utile en cas de multiples remotes).	Nécessite de se souvenir des anciens noms si scripts ou CI l'utilisaient.
<b>git remote remove</b>	<code>git remote remove &lt;nom&gt;</code>	Supprime un dépôt distant de la configuration Git.	Nettoie la liste des remotes inutiles.	Plus aucun push/pull possible vers ce remote sans le recréer.
<b>git status</b>	<code>git status</code>	Affiche l'état des fichiers : suivis, non suivis, prêts à être commit, etc.	Vue rapide et claire des changements avant un commit.	Aucun inconvénient majeur.
<b>git add</b>	<code>git add fichier</code> ou <code>git add .</code>	Ajoute des fichiers (ou tous) dans la zone de staging (préparation au commit).	Contrôle précis sur ce qu'on enregistre.	Fichiers oubliés ne seront pas dans le commit.
<b>git commit</b>	<code>git commit -m "msg"</code>	Valide (sauvegarde) les changements ajoutés dans un nouvel instantané (commit).	Construit l'historique avec un message descriptif.	Si le message n'est pas clair, l'historique sera confus.
<b>git commit -am</b>	<code>git commit -am "msg"</code>	Combine l'ajout et le commit pour les fichiers déjà suivis (pas les nouveaux).	Rapide pour les petites modifications.	Ne prend pas en compte les nouveaux fichiers non suivis.
<b>git commit -amend</b>	<code>git commit --amend</code>	Modifie le dernier commit (message ou fichiers).	Corrige un message ou fichier manquant après coup.	Réécrit l'historique, déconseillé si déjà poussé.

Commande	Utilisation	Explication	Avantage 	Inconvénient 
<b>git log</b>	<code>git log, git log --oneline</code>	Affiche l'historique des commits. Options pour vue détaillée ou simplifiée.	Permet de bien comprendre l'évolution du projet.	Peut être trop verbeux sur gros historiques (sans filtres).
<b>git checkout</b>	<code>git checkout &lt;branche&gt; / git checkout --&lt;fichier&gt;</code>	Change de branche OU annule les changements d'un fichier vers la dernière version committée.	Navigation rapide entre branches.	Risque de perdre des modifs non ajoutées si on change de branche sans stash ou commit.
<b>git branch</b>	<code>git branch &lt;nom&gt;</code>	Crée une nouvelle branche ou liste les branches si utilisé sans argument.	Organisation du travail par fonctionnalités ou correctifs.	Trop de branches non nettoyées peuvent créer la confusion.
<b>git branch -m</b>	<code>git branch -m &lt;ancien&gt; &lt;nouveau&gt;</code>	Renomme une branche locale.	Pratique pour corriger un nom mal choisi.	Peut troubler d'autres utilisateurs si la branche avait déjà été partagée.
<b>git branch -d</b>	<code>git branch -d &lt;nom&gt;</code>	Supprime une branche locale (fusionnée).	Nettoyage des branches terminées.	Impossible si la branche n'a pas été fusionnée (pour éviter de perdre du travail).
<b>git branch -D</b>	<code>git branch -D &lt;nom&gt;</code>	Supprime une branche locale sans vérifier si elle est fusionnée.	Forcer la suppression d'une branche obsolète ou non fusionnée.	Risque de perdre définitivement des commits.
<b>git merge</b>	<code>git merge &lt;branche&gt;</code>	Fusionne la branche spécifiée dans la branche actuelle.	Combine facilement deux flux de travail.	Peut générer des conflits, crée un commit supplémentaire.
<b>git merge --no-ff</b>	<code>git merge --no-ff &lt;branche&gt;</code>	Fusion explicite sans avancer le pointeur (pas de "fast-forward").	Historique plus clair (commit de merge visible).	Ajoute un commit de merge même si un fast-forward était possible.
<b>git rebase</b>	<code>git rebase &lt;branche&gt;</code>	"Rejoue" les commits de la branche actuelle au-dessus de la branche donnée.	Historique linéaire et propre (pas de commit de merge).	Réécrit l'historique (dangereux si la branche est déjà partagée).

Commande	Utilisation	Explication	Avantage 	Inconvénient 
<b>git rebase -i</b>	<code>git rebase -i HEAD~n</code>	Rebase interactif, permet de modifier l'ordre, fusionner ou supprimer des commits récents.	Historique très propre, possibilité de "squasher" plusieurs commits.	Complexe à manipuler, risque de collisions ou de pertes.
<b>git rebase -continue</b>	<code>git rebase --continue</code>	Continue le rebase après avoir résolu un conflit.	Permet de finaliser le rebase.	Faut bien résoudre tous les conflits un par un.
<b>git rebase -abort</b>	<code>git rebase --abort</code>	Annule le rebase en cours et revient à l'état initial.	Permet de stopper si on se retrouve avec trop de conflits.	Annule tout le travail du rebase en cours.
<b>git diff</b>	<code>git diff</code> , <code>git diff --staged</code>	Compare les différences non validées ou en staging.	Affiche précisément les lignes changées.	Peut être difficile à lire sur de gros fichiers ou gros changements.
<b>git blame</b>	<code>git blame &lt;fichier&gt;</code>	Montre, ligne par ligne, qui a modifié le fichier et dans quel commit.	Rapide pour comprendre l'historique d'une ligne.	Peut être moins utile sur un gros fichier avec de multiples modifications.
<b>git bisect</b>	<code>git bisect start</code> , <code>git bisect bad</code> , <code>git bisect good</code>	Recherche binaire d'un commit qui a introduit un bug, en marquant les commits bons/mauvais.	Méthode puissante pour isoler l'origine d'un bug.	Processus un peu long si on ne sait pas exactement où chercher.
<b>git revert</b>	<code>git revert &lt;commit&gt;</code>	Crée un nouveau commit qui annule le commit ciblé sans effacer l'historique.	Historique préservé, sûr pour corriger un commit erroné.	L'historique peut s'allonger avec beaucoup de "reverts".
<b>git reset --soft</b>	<code>git reset --soft HEAD~1</code>	Retour en arrière d'un commit, mais garde les modifications dans la zone de staging.	Permet de refaire un commit propre (re-message).	Réécrit l'historique (dangereux si déjà partagé).
<b>git reset --hard</b>	<code>git reset --hard HEAD~1</code>	Supprime complètement le dernier commit et les modifications associées.	Rapide pour revenir à un état antérieur.	Perte définitive des changements effacés.

Commande	Utilisation	Explication	Avantage 	Inconvénient 
<b>git stash</b>	<code>git stash</code>	Met de côté (sans commit) les modifications en cours (non validées).	Pratique pour basculer sur une autre branche sans perdre son travail.	On peut oublier des stashes s'ils ne sont pas listés ou récupérés.
<b>git stash pop</b>	<code>git stash pop</code>	Récupère le dernier stash et le supprime de la liste des stashes.	Retrouve son travail rapidement pour le continuer.	Peut générer des conflits si le code a beaucoup changé entre-temps.
<b>git stash apply</b>	<code>git stash apply [stash@{n}]</code>	Applique un stash choisi, sans l'enlever de la liste des stashes.	Peut le réappliquer plusieurs fois.	Risque de duplications si on n'est pas prudent.
<b>git stash list</b>	<code>git stash list</code>	Affiche la liste des stashes en attente.	Permet de visualiser tous les travaux mis de côté.	Pas d'inconvénient particulier.
<b>git fetch</b>	<code>git fetch [remote] [branche]</code>	Récupère les nouveautés depuis un dépôt distant, sans fusionner dans la branche locale.	Permet de voir ce qui a changé avant de fusionner.	Demande une étape supplémentaire (pull ou merge) pour intégrer les changements.
<b>git pull</b>	<code>git pull</code>	Récupère et fusionne directement les modifications du dépôt distant dans la branche courante.	Très rapide pour rester à jour.	Peut créer des conflits s'il y a eu des modifications simultanées.
<b>git push</b>	<code>git push</code>	Envoie les commits locaux vers la branche distante correspondante.	Partage facile du travail avec l'équipe.	Peut être refusé s'il y a des conflits ou si l'historique local est en retard.
<b>git push -u</b>	<code>git push -u origin &lt;branche&gt;</code>	Pousse la branche locale et crée un lien de suivi entre la branche locale et distante.	Simplifie les commandes futures (pull/push).	Peut créer de multiples branches sur le remote si mal utilisé.

Commande	Utilisation	Explication	Avantage 	Inconvénient 
<b>git push --force</b>	<code>git push --force</code> ou <code>git push --force-with-lease</code>	Force l'envoi des commits locaux en réécrivant l'historique distant.	Utile pour corriger un rebase ou un amend déjà poussé.	Danger de perdre le travail de collègues (s'il y avait des commits que vous n'avez pas).
<b>git tag</b>	<code>git tag &lt;nom&gt;</code>	Crée un tag léger (non annoté) sur le commit courant.	Pratique pour repérer une version.	Tag léger : pas de description ni auteur de tag.
<b>git tag -a</b>	<code>git tag -a v1.0 -m "message"</code>	Crée un tag annoté (avec message et signature) sur le commit courant.	Complet pour signaler une version stable (historique du tag).	Si trop de tags, peut devenir confus.
<b>git push origin --tags</b>	<code>git push origin --tags</code>	Envoie tous les tags locaux vers le dépôt distant.	Partage public des différentes versions marquées.	Peut propager des tags créés par erreur.
<b>git submodule add</b>	<code>git submodule add &lt;URL&gt;</code>	Ajoute un dépôt comme sous-module dans le projet actuel.	Permet de gérer plusieurs projets liés dans un seul dépôt principal.	Gestion parfois complexe quand les sous-modules changent souvent.
<b>git submodule init</b>	<code>git submodule init</code>	Initialise les sous-modules déjà présents dans le projet.	Indispensable après avoir cloné un projet qui en contient.	Étape supplémentaire qui peut être oubliée.
<b>git submodule update</b>	<code>git submodule update</code>	Met à jour les sous-modules à la version définie dans le dépôt principal.	Synchronise automatiquement avec la référence indiquée par le dépôt principal.	Peut nécessiter un <code>--recursive</code> pour les sous-sous-modules.
<b>git clone --recurse-submodules</b>	<code>git clone --recurse-submodules &lt;URL&gt;</code>	Clone un projet et initialise/mets à jour tous les sous-modules d'un coup.	Très pratique pour récupérer un projet complet.	Sans <code>--recurse-submodules</code> , il faut tout refaire à la main ( <code>init</code> , <code>update</code> ).

Commande	Utilisation	Explication	Avantage 	Inconvénient 
<b>git cherry-pick</b>	<code>git cherry-pick &lt;commit&gt;</code>	Copie un commit précis (d'une autre branche) dans la branche courante.	Permet d'importer une correction ou une fonctionnalité sans fusionner toute la branche.	Peut provoquer des conflits isolés, surtout si le commit est ancien ou dépend d'autres.
<b>git reflog</b>	<code>git reflog</code>	Montre toutes les actions (commits, checkouts, reset...) même ceux qui ne figurent plus dans l'historique normal.	Très utile pour récupérer un commit "perdu" ou en cas de manipulation hasardeuse.	Lecture parfois confuse, surtout pour les débutants.
<b>git gc</b>	<code>git gc</code>	Nettoie et optimise la base de données de Git (compression, suppression d'objets obsolètes).	Peut réduire la taille du dépôt et améliorer les performances.	À utiliser avec précaution (vérifier qu'on n'a pas besoin d'objets récemment supprimés).
<b>git archive</b>	<code>git archive --format=zip HEAD &gt; code.zip</code>	Crée une archive (zip, tar, etc.) du contenu du dépôt à un instant donné (HEAD ou un tag).	Permet d'exporter rapidement une version sans l'historique.	Ne contient pas l'historique Git, juste les fichiers du commit.
<b>git shortlog</b>	<code>git shortlog</code>	Donne un résumé des commits regroupés par auteur.	Vue rapide sur la contribution de chacun.	Manque de détails sur les changements.

#### Remarque :

- Certaines commandes existent avec des variantes ou options supplémentaires (ex. `-m`, `-p`, `--stat`).
- Chaque commande peut devenir plus puissante avec ces options, mais la logique globale reste la même.
- Les avantages et inconvénients dépendent aussi du contexte (travail solo, équipe, taille du projet, etc.).

Ce tableau synthétise l'essentiel, en classant les commandes par leur nom et en indiquant leur usage principal, un bref résumé, ainsi que leurs atouts et limites.

## Conclusion

Avec ce cours détaillé, vous avez un **aperçu complet** des fonctionnalités de Git, depuis les **fondamentaux** (init, clone, commit) jusqu'aux **concepts avancés** (rebase, cherry-pick, submodules).

En pratiquant régulièrement, vous acquerrez progressivement les bons réflexes et saurez tirer le meilleur parti de Git et de ses puissantes capacités de gestion de versions. N'hésitez pas à revenir à ce document



en cas de doute et à expérimenter sur de petits projets pour mieux apprivoiser chaque commande.

**Bonne utilisation de Git et bonne continuation dans vos projets !**