Git et GitHub

Concepts et commandes fondamentales et avancées

Luca Italo Mansutti

luca.mansutti7@gmail.com

Table des matières

[Partie 1 : Concepts Fondamentaux 1](#_Toc183387566)

[1 Configuration de Git 1](#_Toc183387567)

[1.1 Niveaux de configurations 1](#_Toc183387568)

[1.2 Ajouter des configurations 1](#_Toc183387569)

[1.3 Supprimer des configurations 1](#_Toc183387570)

[1.3.1 Une clé spécifique 1](#_Toc183387571)

[1.3.2 Toutes les occurrences d’une clé 1](#_Toc183387572)

[1.3.3 Une section entière 1](#_Toc183387573)

[1.4 Consulter les configurations 2](#_Toc183387574)

[1.4.1 Globales 2](#_Toc183387575)

[1.4.2 Locales 2](#_Toc183387576)

[1.4.3 Valeur unique 2](#_Toc183387577)

[1.5 Configurations spécifiques 2](#_Toc183387578)

[1.5.1 S’identifier à git globalement 2](#_Toc183387579)

[1.5.2 Changer le nom de la branche par défaut 2](#_Toc183387580)

[1.5.3 Choisir l’éditeur par défaut 2](#_Toc183387581)

[1.5.4 Choisir entre merge/rebase lors d’un pull 2](#_Toc183387582)

[1.5.5 Activer ou non le RERERE (voir chapitre 17) 2](#_Toc183387583)

[2 Repository 3](#_Toc183387584)

[2.1 Initialisation du repository 3](#_Toc183387585)

[2.1.1 Créer un dossier et des fichiers 3](#_Toc183387586)

[2.1.2 Initialiser le dossier comme dépôt Git 3](#_Toc183387587)

[2.2 Types d’états de fichiers 3](#_Toc183387588)

[2.3 Status 3](#_Toc183387589)

[2.4 Stages 3](#_Toc183387590)

[2.4.1 Ajouter un fichier au staging 3](#_Toc183387591)

[2.4.2 Retirer du staging et arrêter de suivre le fichier 3](#_Toc183387592)

[2.4.3 Annuler un ajout au staging 3](#_Toc183387593)

[2.5 Commit 4](#_Toc183387594)

[2.5.1 Faire un commit avec un message 4](#_Toc183387595)

[2.5.2 Modifier le message du dernier commit en cas d’erreur 4](#_Toc183387596)

[3 Git log 4](#_Toc183387597)

[3.1 Informations fournies par git log 4](#_Toc183387598)

[3.2 Log command 4](#_Toc183387599)

[3.3 Options (flags) de la commande git log 4](#_Toc183387600)

[3.3.1 Limiter l’affichage à un certain nombre de commits 4](#_Toc183387601)

[3.3.2 Affichage simplifié en une ligne par commit 4](#_Toc183387602)

[3.3.3 Afficher l’historique de toutes les branches (pour voir les merges) 5](#_Toc183387603)

[3.3.4 Affichage graphique des commits 5](#_Toc183387604)

[3.3.5 Afficher les commits parents 5](#_Toc183387605)

[3.3.6 Afficher les commits d'une branche distante spécifique 5](#_Toc183387606)

[3.3.7 Afficher les informations des fichiers modifiés des commits 5](#_Toc183387607)

[3.3.8 Afficher les différences entre tous les commits 5](#_Toc183387608)

[3.3.9 Afficher toutes les références (pointeurs vers les commits) 5](#_Toc183387609)

[3.3.10 Limiter les logs à certaines dates 5](#_Toc183387610)

[3.3.11 Afficher les commits d’un auteur en particulier 5](#_Toc183387611)

[4 Fonctionnement interne 6](#_Toc183387612)

[4.1 Bases du fonctionnement interne (Plumbing) 6](#_Toc183387613)

[4.2 Exploration des objets 6](#_Toc183387614)

[4.2.1 Trouver le hash d’un commit 6](#_Toc183387615)

[4.2.2 Lister le contenu du dossier .git/objects 6](#_Toc183387616)

[4.3 Commande cat-file 6](#_Toc183387617)

[4.3.1 Afficher le contenu d’un commit 6](#_Toc183387618)

[4.4 Trees & Blobs 7](#_Toc183387619)

[4.4.1 Exploration d’un objet commit 7](#_Toc183387620)

[4.4.2 Examiner un objet Tree 7](#_Toc183387621)

[4.4.3 Examiner un objet Blob 7](#_Toc183387622)

[4.5 Commits multiples (parents) 8](#_Toc183387623)

[4.6 Stockage des données 8](#_Toc183387624)

[4.6.1 Comment git stocke les données 8](#_Toc183387625)

[4.6.2 Optimisation du stockage 8](#_Toc183387626)

[5 Branching 9](#_Toc183387627)

[5.1 Utilité des branches 9](#_Toc183387628)

[5.2 Fonctionnement interne 9](#_Toc183387629)

[5.3 Opérations sur les branches 10](#_Toc183387630)

[5.3.1 Afficher toutes les branches du repository 10](#_Toc183387631)

[5.3.2 Créer une branche sans y basculer directement 10](#_Toc183387632)

[5.3.3 Créer et basculer directement sur la nouvelle branche 10](#_Toc183387633)

[5.3.4 Basculer entre les branches 10](#_Toc183387634)

[5.3.5 Renommer une branche 10](#_Toc183387635)

[5.3.6 Supprimer une branche 10](#_Toc183387636)

[5.4 Ajout d’un commit dans une nouvelle branche 10](#_Toc183387637)

[5.5 Le fichier git et les branches 11](#_Toc183387638)

[5.5.1 Branches et leurs références 11](#_Toc183387639)

[5.5.2 Commandes pour examiner les références 11](#_Toc183387640)

[5.5.2.1 Afficher le hash du commit d'une branche 11](#_Toc183387641)

[5.5.2.2 Afficher toutes les branches 11](#_Toc183387642)

[5.5.2.3 Afficher toutes les branches et leur commits 11](#_Toc183387643)

[6 Merge 12](#_Toc183387644)

[6.1 Pourquoi avoir plusieurs branches ? 12](#_Toc183387645)

[6.2 Merge command 12](#_Toc183387646)

[6.3 Merge commits 12](#_Toc183387647)

[6.3.1 Fonctionnement des merge commits 12](#_Toc183387648)

[6.4 Merge log 13](#_Toc183387649)

[6.4.1 Commande de vérification du merge 13](#_Toc183387650)

[6.4.2 Explications des hash avec des parents 13](#_Toc183387651)

[6.4.2.1 Commit de fusion (Merge Commit) 13](#_Toc183387652)

[6.4.2.2 Commits de la branche add\_classics 13](#_Toc183387653)

[6.4.2.3 Commits de la branche main : 13](#_Toc183387654)

[6.5 Fast forward merge / commit 14](#_Toc183387655)

[7 Rebase 15](#_Toc183387656)

[7.1 Visualisation d’un rebase 15](#_Toc183387657)

[7.2 Exécuter un rebase 15](#_Toc183387658)

[7.2.1 Se positionner dans la branche dans laquelle on veut ramener les commits 15](#_Toc183387659)

[7.2.2 Commande Rebase 15](#_Toc183387660)

[7.3 Merge vs Rebase 16](#_Toc183387661)

[7.3.1 Merge 16](#_Toc183387662)

[7.3.2 Rebase : 16](#_Toc183387663)

[7.4 Quand utiliser rebase ? 16](#_Toc183387664)

[7.5 Quand éviter rebase ? 16](#_Toc183387665)

[8 Annulation et restauration 17](#_Toc183387666)

[8.1 Reset Command 17](#_Toc183387667)

[8.1.1 Soft reset 17](#_Toc183387668)

[8.1.2 Hard reset 17](#_Toc183387669)

[17](#_Toc183387670)

[8.1.3 Dangers du hard reset 17](#_Toc183387671)

[8.2 Restore command 18](#_Toc183387672)

[8.2.1 Restaurer des fichiers du dernier commit dans le répertoire local 18](#_Toc183387673)

[8.2.2 Annuler un ajout au stage (unstage) : 18](#_Toc183387674)

[8.3 Revert Command 18](#_Toc183387675)

[8.4 Récapitulatif 18](#_Toc183387676)

[9 Remote 19](#_Toc183387677)

[9.1 Gérer des dépôts distants 19](#_Toc183387678)

[9.2 Ajouter un remote 19](#_Toc183387679)

[9.3 Fetch 19](#_Toc183387680)

[9.3.1 Fetch command 19](#_Toc183387681)

[9.3.2 Fetch n’intègre pas les commits du dépôt distant 19](#_Toc183387682)

[9.3.3 Conclusion 19](#_Toc183387683)

[9.4 Log remote 20](#_Toc183387684)

[9.4.1 Afficher l'Historique d'une Branche Distante avec git log 20](#_Toc183387685)

[9.5 Merge Remote 20](#_Toc183387686)

[9.5.1 Fusionner une Branche Distante dans une Branche Locale 20](#_Toc183387687)

[9.5.2 Merge remote command 20](#_Toc183387688)

[10 GitHub 21](#_Toc183387689)

[10.1 GitHub repository 21](#_Toc183387690)

[10.1.1 GitHub 21](#_Toc183387691)

[10.1.2 GitHub CLI 21](#_Toc183387692)

[10.2 Gestion du dépôt sur GitHub 22](#_Toc183387693)

[10.2.1 Créer un dépôt sur GitHub 22](#_Toc183387694)

[10.2.2 S’authentifier avec GitHub CLI (gh) 22](#_Toc183387695)

[10.3 Remote add command 23](#_Toc183387696)

[10.3.1 Lier le dépôt local au dépot distant (remote) sur GitHub 23](#_Toc183387697)

[10.3.2 Vérifier la liaison 23](#_Toc183387698)

[10.4 Git Push 23](#_Toc183387699)

[10.4.1 Options alternatives 23](#_Toc183387700)

[10.5 Git Pull 23](#_Toc183387701)

[10.6 Pull request 24](#_Toc183387702)

[10.6.1 Fonctionnement d’un pull request 24](#_Toc183387703)

[10.6.2 Exemple de pull request 24](#_Toc183387704)

[10.6.2.1 Partie "Envoyer les modifications" 24](#_Toc183387705)

[10.6.2.2 Partie "Création d'une Pull Request" 25](#_Toc183387706)

[11 GitIgnore 26](#_Toc183387707)

[11.1 Fichier .gitignore 26](#_Toc183387708)

[11.1.1 Exemple d’utilisation 26](#_Toc183387709)

[11.2 Nested .gitignore 26](#_Toc183387710)

[11.2.1 Fonctionnement 26](#_Toc183387711)

[11.2.2 Exemple concret 27](#_Toc183387712)

[11.3 Patterns 27](#_Toc183387713)

[11.3.1 Wildcard (\*) 27](#_Toc183387714)

[11.3.2 Double wildcard (\*\*) 27](#_Toc183387715)

[11.3.3 Rooted patterns (/) 27](#_Toc183387716)

[11.3.4 Négation 28](#_Toc183387717)

[11.3.5 Commentaires 28](#_Toc183387718)

[11.4 Que mettre dans le .gitignore ? 28](#_Toc183387719)

[11.4.1 Fichiers générés 28](#_Toc183387720)

[11.4.2 Dépendances 28](#_Toc183387721)

[11.4.3 Fichiers personnels ou spécifiques à l'éditeur 29](#_Toc183387722)

[11.4.4 Informations sensibles ou dangereuses 29](#_Toc183387723)

[12 README.md 30](#_Toc183387724)

[12.1 Qu’est-ce QU’UN fichier README ? 30](#_Toc183387725)

[12.2 Syntaxe d’un fichier .md 30](#_Toc183387726)

[12.2.1 Titres 30](#_Toc183387727)

[12.2.2 Gras et italique 30](#_Toc183387728)

[12.2.3 Listes 30](#_Toc183387729)

[12.2.4 Liens et images 31](#_Toc183387730)

[12.2.5 Bloc de code avec coloration syntaxique 31](#_Toc183387731)

[12.2.6 Tableaux 31](#_Toc183387732)

[12.2.7 Saut de ligne 31](#_Toc183387733)

[12.2.8 Citations 31](#_Toc183387734)

[12.2.9 Séquences d’échappement 31](#_Toc183387735)

[12.3 Exemple complet de fichier .md 32](#_Toc183387736)

[12.3.1 Code markdown 32](#_Toc183387737)

[12.3.2 Affichage sur GitHub 33](#_Toc183387738)

[12.4 Fichier readme de profil GitHub 33](#_Toc183387739)

[12.4.1 Marche à suivre 33](#_Toc183387740)

[12.4.2 Exemple de présentation de profil GitHub 34](#_Toc183387741)

[Partie 2 : Concepts avancés 1](#_Toc183387742)

[13 fork 35](#_Toc183387743)

[13.1 Qu’est-ce qu’un fork ? 35](#_Toc183387744)

[13.2 Comment fork un repository 35](#_Toc183387745)

[13.3 Exemple réel de travail collaboratif 35](#_Toc183387746)

[13.3.1 Cloner le dépôt "forké" 35](#_Toc183387747)

[13.3.2 Créer une nouvelle branche pour les nouvelles fonctionnalités 35](#_Toc183387748)

[13.3.3 Ajouter et commit les modifications 35](#_Toc183387749)

[13.3.4 Pousser les changements vers GitHub 35](#_Toc183387750)

[13.3.5 Créer un pull request 35](#_Toc183387751)

[14 Reflog 36](#_Toc183387752)

[14.1 Head 36](#_Toc183387753)

[14.1.1 Voir où pointe HEAD 36](#_Toc183387754)

[14.2 Reflog command 36](#_Toc183387755)

[14.2.1 Explications de reflog 36](#_Toc183387756)

[14.2.2 Fonctionnement de reflog 36](#_Toc183387757)

[14.2.3 Log VS Reflog 36](#_Toc183387758)

[14.3 Récupération du contenu d’un fichier supprimé 37](#_Toc183387759)

[14.3.1 Étapes à suivre pour récupérer le contenu d’un fichier supprimé 37](#_Toc183387760)

[14.3.1.1 Vérifier l’historique avec reflog 37](#_Toc183387761)

[14.3.1.2 Explorer le commit avec git cat-file -p 37](#_Toc183387762)

[14.3.1.3 Explorer le tree 37](#_Toc183387763)

[14.3.1.4 Examiner le blob pour retrouver le contenu du fichier 37](#_Toc183387764)

[14.4 Récupération améliorée avec merge 38](#_Toc183387765)

[14.4.1 Merge avec un commitish 38](#_Toc183387766)

[15 Merge conflicts 39](#_Toc183387767)

[15.1 Introduction aux conflits 39](#_Toc183387768)

[15.1.1 Exemple de conflit 39](#_Toc183387769)

[15.2 Résoudre un conflit 40](#_Toc183387770)

[15.2.1 Localiser le conflit 40](#_Toc183387771)

[15.2.2 Éditer les fichier et résoudre les conflits 40](#_Toc183387772)

[15.2.3 Ajouter et finaliser la fusion 40](#_Toc183387773)

[15.2.4 Vérifier le merge 40](#_Toc183387774)

[15.3 Arrêter un merge lors d’un conflit 41](#_Toc183387775)

[15.4 Outils de résolution de conflits intégrée à Git 41](#_Toc183387776)

[15.4.1 "Ours" vs "Theirs" (Notre vs Leurs) 41](#_Toc183387777)

[15.4.2 Checkout merge command 41](#_Toc183387778)

[16 Rebase conflicts 42](#_Toc183387779)

[16.1 Exemple de rebase conflicts 42](#_Toc183387780)

[16.1.1 Scénario : 42](#_Toc183387781)

[16.1.2 L'état "detached HEAD" 42](#_Toc183387782)

[16.1.3 Checkout rebase command 43](#_Toc183387783)

[16.1.4 Terminer le rebase 43](#_Toc183387784)

[17 RERERE (Reuse recorded resolution) 44](#_Toc183387785)

[17.1 Qu’est-ce que git rerere 44](#_Toc183387786)

[17.1.1 Activer / désactiver rerere 44](#_Toc183387787)

[17.1.2 Supprimer le dossier rerere 44](#_Toc183387788)

[17.2 Exemple d’utilisation de rerere 44](#_Toc183387789)

[17.2.1 Premier rebase de main sur favs (Entrée dans la zone de conflit) 44](#_Toc183387790)

[17.2.2 Résolution des conflits et fermeture du rebase 45](#_Toc183387791)

[17.2.3 Rebase de main sur favs2 (Résolution automatique) 45](#_Toc183387792)

[17.2.4 Fermeture du rebase 45](#_Toc183387793)

[17.3 Annuler un commit (alors qu’il fallait --continue) 45](#_Toc183387794)

[17.3.1 Problème courant 45](#_Toc183387795)

[17.3.2 Annuler un commit accidentel 45](#_Toc183387796)

[18 Squash 46](#_Toc183387797)

[18.1 Qu’est-ce que le squashing ? 46](#_Toc183387798)

[18.2 Comment faire un squash ? 46](#_Toc183387799)

[18.2.1 Démarrer un rebase interactif 46](#_Toc183387800)

[18.2.2 Choisir les commits à squash 46](#_Toc183387801)

[18.3 Overwrite main 47](#_Toc183387802)

[18.3.1 Supprimer la branche main 47](#_Toc183387803)

[18.3.2 Renommer la branche temporaire par main 47](#_Toc183387804)

[18.4 Dangers du squashing 47](#_Toc183387805)

[18.5 Squashing pull requests 48](#_Toc183387806)

[18.5.1 Résumé du flux de travail 48](#_Toc183387807)

[18.6 Force push 48](#_Toc183387808)

[18.6.1 Qu’est-ce qu’un force push 48](#_Toc183387809)

[18.6.2 Force push command 48](#_Toc183387810)

[19 Stash 49](#_Toc183387811)

[19.1 A quoi sert le stashing ? 49](#_Toc183387812)

[19.2 Stash command 49](#_Toc183387813)

[19.2.1 Créer un stash 49](#_Toc183387814)

[19.2.2 Lister les stashes 49](#_Toc183387815)

[19.2.3 Stash Pop 49](#_Toc183387816)

[19.2.4 Stash Apply 49](#_Toc183387817)

[19.2.5 Stash drop 49](#_Toc183387818)

[20 Diff 50](#_Toc183387819)

[20.1 Diff command 50](#_Toc183387820)

[20.1.1 Comparer l’état actuel et le dernier commit 50](#_Toc183387821)

[20.1.2 Comparer un commit précédent et l'état actuel 50](#_Toc183387822)

[20.1.3 Comparer deux commits spécifiques 50](#_Toc183387823)

[20.1.4 Comparer l'index (zone de staging) avec la dernière validation 50](#_Toc183387824)

[20.1.5 Comparer l'arbre de travail avec l'index 50](#_Toc183387825)

[Exemple d’utilisation de diff 50](#_Toc183387826)

[20.2 Blame command 51](#_Toc183387827)

[20.2.1 Exemple de sortie de git blame 51](#_Toc183387828)

[21 Cherry pick 52](#_Toc183387829)

[21.1 Que fait la commande cherry-pick ? 52](#_Toc183387830)

[21.2 Cherry-pick command 52](#_Toc183387831)

[22 Bisect 53](#_Toc183387832)

[22.1 A quoi sert le bisect ? 53](#_Toc183387833)

[22.2 Marche à suivre du bisect 53](#_Toc183387834)

[22.2.1 Commencer le bisect 53](#_Toc183387835)

[22.2.2 Marquer un commit fonctionnel (bon commit) 53](#_Toc183387836)

[22.2.3 Marquer un commit problématique (mauvais commit) 53](#_Toc183387837)

[22.2.4 Tester le commit intermédiaire 53](#_Toc183387838)

[22.2.5 Déclarer le résultat du test 53](#_Toc183387839)

[22.2.6 Répéter les 2 étapes précédentes 53](#_Toc183387840)

[22.2.7 Terminer la session de bisect 53](#_Toc183387841)

[23 Worktree 54](#_Toc183387842)

[23.1 Qu’est-ce qu’un worktree ? 54](#_Toc183387843)

[23.2 Types de worktrees 54](#_Toc183387844)

[23.2.1 Main Worktree 54](#_Toc183387845)

[23.2.2 Linked Worktree 54](#_Toc183387846)

[23.3 Worktree commands 54](#_Toc183387847)

[23.3.1 Lister tous les worktrees associés à un repository 54](#_Toc183387848)

[23.3.2 Créer un linked worktree 54](#_Toc183387849)

[23.3.3 Afficher le contenu du fichier .git d’un linked worktree 54](#_Toc183387850)

[23.4 pas de branches partagées entre worktrees ! 55](#_Toc183387851)

[23.5 Tracking des linked worktrees 55](#_Toc183387852)

[23.6 Main et linked worktrees : Système unique 55](#_Toc183387853)

[23.7 Supprimer un worktree 56](#_Toc183387854)

[23.7.1 Supprimer un worktree ne supprime pas la branche 56](#_Toc183387855)

[23.7.2 Remove command 56](#_Toc183387856)

[23.7.3 Prune command 56](#_Toc183387857)

[24 Tags 57](#_Toc183387858)

[24.1 Qu’est-ce qu’un tag ? 57](#_Toc183387859)

[24.2 Tag commands 57](#_Toc183387860)

[24.2.1 Lister les tags existants 57](#_Toc183387861)

[24.2.1.1 Localement 57](#_Toc183387862)

[24.2.1.2 Sur un dépôt distant 57](#_Toc183387863)

[Afficher les détails d’un tag 57](#_Toc183387864)

[24.2.2 Créer un tag annoté 57](#_Toc183387865)

[24.2.3 Supprimer un tag 57](#_Toc183387866)

[24.2.3.1 Localement 57](#_Toc183387867)

[24.2.3.2 Sur le dépôt distant 57](#_Toc183387868)

[24.2.4 Pousser les tags vers le dépôt distant 57](#_Toc183387869)

[Formulaire des commandes 30](#_Toc183387870)

[25 Commandes fondamentales 57](#_Toc183387871)

[26 Commandes avancées 60](#_Toc183387872)

[Sources 61](#_Toc183387873)

[27 Références 62](#_Toc183387874)

Une image contenant conception

Description générée automatiquement avec une confiance faible

Partie 1 : Concepts Fondamentaux

Une image contenant symbole, clipart, créativité

Description générée automatiquement

# Configuration de Git

## Niveaux de configurations

Les configurations sont hiérarchisées, du plus général au plus spécifique :

1. Une image contenant texte, cercle, Police

   Description générée automatiquement**Système** : /etc/gitconfig  
   Impacte tous les utilisateurs du système.
2. **Global** : ~/.gitconfig  
   Impacte tous les dépôts de l’utilisateur.
3. **Local** : .git/config  
   Impacte uniquement le dépôt courant.
4. **Worktree** : .git/config.worktree  
   Spécifique à une branche ou partie d’un dépôt.

**Remarque** : Les configurations **locales écrasent les configurations globales**

**worktree > local > global > system**

## Ajouter des configurations

**git config** [--add] {–-global | --local} <SECTION>.<KEYNAME> "VALEUR"

**Exemple** : **git config** --add --global netflix.ceo "Luca Mansutti”

## Supprimer des configurations

### Une clé spécifique

**git config** -–unset [--global] <SECTION>.<KEYNAME>

### Toutes les occurrences d’une clé

**git config** –-unset-all [--global] <SECTION>.<KEYNAME>

### Une section entière

**git config** --remove-section [--global] <SECTION>

## Consulter les configurations

### Globales

**git config** --list # Toutes les configurations globales

### Locales

**git config** --list --local # Configurations locales uniquement

**cat** .git/config # Affiche les configurations locales

### Valeur unique

**git config** --get <SECTION>.<KEYNAME> # Valeur associée à la clé

## Configurations spécifiques

### S’identifier à git globalement

**git config** [--add] –-global user.name "USERNAME"

**git config** [--add] --global user.email "EMAIL"

### Changer le nom de la branche par défaut

**git config** [--add] –-global init.defaultBranch <BRANCH\_NAME>

### Choisir l’éditeur par défaut

**git config** –-global core.editor "EDITOR\_NAME"

### Choisir si git prend en compte la casse

**git config** –-global core.ignorecase [TRUE | FALSE]

**Note** : De base Git ne prend pas en compte la casse dans les fichiers, c’est embêtant car les changements de nom de fichiers (min/maj) ne sont pas pris en compte, on peut donc lui dire d’ignorer la casse :

### Choisir entre merge/rebase lors d’un pull

Si vous préférez que git pull utilise le rebase au lieu de merge :

**git config** --global pull.rebase [TRUE | FALSE]

### Activer ou non le RERERE (voir chapitre 17)

**git config** --global rerere.enabled true

# Repository

## Initialisation du repository

### Créer un dossier et des fichiers

**mkdir** <NOM\_DU\_DOSSIER>

**cd** <NOM\_DU\_DOSSIER/>

**touch** <NOM\_DU\_FICHIER>

### Initialiser le dossier comme dépôt Git

git init

**Note :** Le dossier .git est créé pour contenir toutes les informations de suivi. Supprimer ce dossier arrête la liaison avec Git

## Types d’états de fichiers

* **untracked :** non suivis par git (ignoré)
* **staged :** prêt à être commit (en attente)
* **committed :** déjà commit (sauvé dans l’historique des commit du repo)

## Status

**git status**

**Note :** Cette commande permet d’afficher l’état des fichiers d’un repository

## Stages

### Ajouter un fichier au staging

**git add** <PATH | PATTERN | . | FILE>

**Note :** Cette commande permet de choisir les fichiers à inclure pour le prochain commit

### Retirer du staging et arrêter de suivre le fichier

**git rm** --cached <FILE>

**Note** : Le fichier ne sera plus suivi par Git

### Annuler un ajout au staging

**git restore** --staged <FILE>

**Note** : Le fichier reste suivi par Git

## Commit

### Faire un commit avec un message

**git commit** -m "MESSAGE"

**Note :** Le commit crée une snapshot du repository à un moment donné (historique). Cela crée une sauvegarde des fichiers à ce moment-là.

Si les fichiers sont déjà suivis, on peut faire un commit et un add en une seule ligne :

**git commit** -am "MESSAGE"

### Modifier le message du dernier commit en cas d’erreur

**git commit** –-amend -m "MESSAGE"

# Git log

## Informations fournies par git log

La commande git log affiche l'historique des commits du projet

* **Auteur** : Qui a réalisé le commit.
* **Date** : Quand le commit a été effectué.
* **Modifications** : Les fichiers ou parties modifiées.
* **Commit hash** : Un identifiant unique (longue chaîne de caractères) qui identifie le commit, par exemple : 5ba786fcc93e8092831c01e71444b9baa2228a4f

**Note** : Pour simplifier, il suffit d’utiliser les 7 premiers caractères du hash (ex. : 5ba786f).

## Log command

**git log** [OPTIONS]

**Note** : La commande démarre dans un **pager interactif** qui permet de naviguer avec les flèches du clavier et de quitter avec q..

## Options (flags) de la commande git log

### Limiter l’affichage à un certain nombre de commits

-n <NOMBRE>

### Affichage simplifié en une ligne par commit

--oneline

### Afficher l’historique de toutes les branches (pour voir les merges)

--all

### Affichage graphique des commits

--graph

### Afficher les commits parents

--parents

### Afficher les commits d'une branche distante spécifique

<REMOTE>/<BRANCH>

### Afficher les informations des fichiers modifiés des commits

--stat

### Afficher les différences entre tous les commits

-p

### Afficher toutes les références (pointeurs vers les commits)

--decorate=<FULL | SHORT | NO>

* short (par défaut) : affiche les noms abrégés des références.
* full : affiche le nom complet des références.
* no : n’affiche pas les références.

### Limiter les logs à certaines dates

**git log** --since=”date” --until=”date”

* Dates relatives : “1.week.ago”, “yesterday”, “2.days.ago”
* Dates absolues : "2024-11-23", "2024-11-23T15:30:00"
* Expressions naturelles : “last Friday”, “1 month ago”

### Afficher les commits d’un auteur en particulier

**git log** --author="<nom\_ou\_email>"

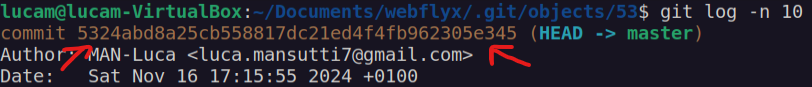
# Fonctionnement interne

## Bases du fonctionnement interne (Plumbing)

L'intégralité des données d'un dépôt Git (commits, branches, tags, objets, etc.) est stockée dans le dossier caché .git. Les objets Git sont enregistrés dans le répertoire **.git/objects**/ où chaque commit n’est qu’un type d’objet parmi d’autres.

## Exploration des objets

### Trouver le hash d’un commit

**git log**

**Note** : Le but est de trouver les 2 premiers digits du hash et de retrouver l’objet correspondant

### Lister le contenu du dossier .git/objects

**Une image contenant texte, capture d’écran, Police

Description générée automatiquementls** -al .git/objects

## Commande cat-file

### Afficher le contenu d’un commit

**cat-file** est une commande de git qui permet de lire le contenu d’un commit

**git cat-file** -p <COMMIT\_HASH>

Une image contenant texte, capture d’écran, Police

Description générée automatiquement**Note** : Mettre le hash complet ou les 7 premiers digits. Le -p (pretty print) affiche le contenu d’une manière lisible

## Trees & Blobs

Dans Git, les **arbres** (trees) et les **blobs** sont des objets qui servent à structurer et stocker les données du dépôt.

* **Tree** : Représente un **répertoire** (ou un dossier). Il contient des références vers des blobs (fichiers) ou d’autres arbres (sous-répertoires).
* **Blob** : Représente le contenu d’un **fichier**. C’est un objet de données pur sans structure hiérarchique.

### Exploration d’un objet commit

**git cat-file** -p <COMMIT\_HASH>

**Une image contenant texte, capture d’écran, Police

Description générée automatiquementRésultat** : Cela t'affiche les informations sur le commit (auteur, date, message du commit) ainsi que l’objet **tree** associé, qui représente la structure des fichiers et répertoires du commit.

### Examiner un objet Tree

**git cat-file** -p <TREE\_HASH>

**Résultat** : Cela te montrera une liste de fichiers et sous-répertoires associés à ce commit. Chaque fichier est représenté par un objet **blob** avec un hash unique.



### Examiner un objet Blob

**git cat-file** -p <BLOB\_HASH>

**Résultat** : Cela affiche le **contenu du fichier** sous forme de texte (si le fichier est lisible).

## Commits multiples (parents)

Une image contenant texte, capture d’écran, Police

Description générée automatiquementAvec la commande git log nous pouvons voir le nouveau commit effectué

Une image contenant texte, capture d’écran, Police

Description générée automatiquementAvec la commande git cat-file -p <COMMIT\_HASH> nous voyons désormais le paramètre **parent** (le commit précédent dans l’historique du dépôt)

## Stockage des données

### Comment git stocke les données

Git stocke une **photo instantanée** (snapshot) de tous les fichiers à chaque commit, ce qui signifie que chaque commit contient une version complète des fichiers. Contrairement à ce que l'on pourrait penser, Git ne stocke pas seulement les changements effectués dans un commit, mais plutôt l'état complet des fichiers.

### Optimisation du stockage

Bien que Git stocke des snapshots complets, il optimise le stockage afin de ne pas rendre le répertoire .git trop volumineux :

* Git **compresse** et **regroupe** les fichiers pour les stocker de manière plus efficace.
* Git **déduplique** les fichiers identiques entre les commits : si un fichier n'a pas changé entre deux commits, Git ne le stocke qu'une seule fois.

Les fichiers qui n’ont pas changé entre 2 commits possèdent le même hash\_blob

# Branching

## Utilité des branches

En Git, une **branche** est un mécanisme qui permet de suivre des modifications séparément, sans affecter directement le projet principal.

Deux scénarios possibles :

* **Les modifications sont satisfaisantes :** Vous fusionnez (**merge**) la branche X avec la branche principale (master ou main).
* **Les modifications ne conviennent pas :** Vous supprimez la branche X sans que la branche principale soit affectée.

## Fonctionnement interne

**Une branche = un pointeur nommé vers un commit spécifique.**  
Lorsque vous créez une branche, Git crée simplement un "pointeur" vers un commit existant.

**La branche suit les nouveaux commits.**  
À chaque nouveau commit, le pointeur de la branche se déplace automatiquement pour désigner le commit le plus récent (appelé "**tip** de la branche").

**Branches légères et efficaces :**

* Les branches ne dupliquent pas le projet sur le disque dur.
* Créer plusieurs branches est rapide et peu gourmand en ressources.

Une image contenant diagramme, capture d’écran, ligne

Description générée automatiquement

## Opérations sur les branches

### Afficher toutes les branches du repository

**git branch** [--list]

**Note** : la branche avec le \* est la branche dans laquelle on se trouve

### Créer une branche sans y basculer directement

**git branch** <BRANCH> [<COMMIT\_HASH>]

**Note** : le flag COMMIT\_HASH permet de choisir à partir de quel commit on veut créer la branche (à utiliser si on ne veut pas créer la branche à partir du dernier commit)

### Créer et basculer directement sur la nouvelle branche

**git switch** -c <BRANCH> [<COMMIT\_HASH>]

### Basculer entre les branches

**git switch** <BRANCH>

### Renommer une branche

**git branch** -m <OLD\_NAME> <NEW\_NAME>

**Note** : GitHub utilise main pour la branche pas défaut

Ex : git branch -m master main cela change la branche "master" en "main"

### Supprimer une branche

**git branch** -d <BRANCH>

## Ajout d’un commit dans une nouvelle branche

**Exemple** : Ajout de la branche "add\_classics", contenant un nouveau fichier "classics.csv"

git switch -c add\_classics

touch classics.csv

git add .

git commit -m "Ajout de classics.csv"

Une image contenant texte, capture d’écran, Police

Description générée automatiquementVérifions le résultat avec git log :

Résultat attendu :

* La branche add\_classics contient un commit supplémentaire (D).
* Une image contenant texte, Police, blanc, conception

  Description générée automatiquementLa branche main reste intacte, pointant toujours au commit C.

## Le fichier git et les branches

Git stocke toutes les informations liées à votre projet dans le répertoire caché .git à la racine de votre projet. Ce répertoire contient des sous-dossiers et fichiers organisés pour suivre les commits, branches, objets et configurations.

### Branches et leurs références

Les branches sont représentées comme des fichiers dans le répertoire .git/refs/heads.

Chaque fichier porte le nom d'une branche et contient le hash du commit vers lequel pointe cette branche.

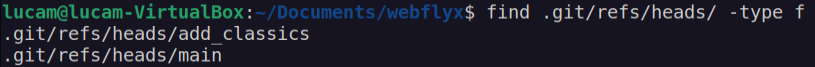
**Exemple :** Le **fichier .git/refs/heads/main** contient le hash du dernier commit

### Commandes pour examiner les références

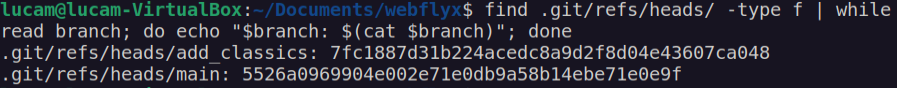
#### Afficher le hash du commit d'une branche

**cat** .git/refs/heads/<BRANCH>

#### Afficher toutes les branches

**find** .git/refs/heads -type f

#### Afficher toutes les branches et leur commits

**find** .git/refs/heads -type f | while read branch; do echo "$branch: $(cat $branch)"; done

# Merge

## Pourquoi avoir plusieurs branches ?

Une image contenant texte, Police, blanc, conception

Description générée automatiquementLes branches multiples permettent de faire des modifications sans affecter la branche principale. Une fois les modifications terminées, vous pouvez les fusionner avec la branche principale pour les intégrer au produit final.

**Exemple :** Supposons que vous ayez deux branches avec des commits uniques :

Une image contenant texte, Police, diagramme, blanc

Description générée automatiquementSi vous fusionnez **other\_branch** dans **main**, Git combine les deux branches en créant un nouveau commit de fusion (F) qui a à la fois C et E comme parents :

Dans cet exemple, F ramène les modifications de D et E dans la branche principale.

## Merge command

**git merge** <OTHER\_BRANCH> -m "MESSAGE"

**Note** : Avant d’effectuer cette commande se redéplacer dans la branche main (ou autre)

## Merge commits

### Fonctionnement des merge commits

Un **merge commit** est un commit spécial créé lorsqu'on fusionne deux branches.

Une image contenant texte, Police, blanc, conception

Description générée automatiquement**Exemple** : Imaginons que nous avons deux branches, **main** et **add\_classics**, avec leurs commits respectifs

Fusionnons la branche add\_classics dans main

🡪 Git va procéder de la manière suivante :

* **Trouver le commit de base de fusion** : C'est le commit **A**, l'ancêtre commun des deux branches.
* **Rejouer les changements de main** (depuis le commit de base, donc depuis A).
* **Rejouer les changements de add\_classics** (depuis le commit de base, donc depuis A).
* Une image contenant texte, Police, blanc, diagramme

  Description générée automatiquement**Créer un commit de fusion** (F), qui aura **deux parents** : **C** (de main) et **E** (de add\_classics).

## Merge log

### Commande de vérification du merge

**git log** --oneline --graph --parents [--all]

Une image contenant texte, capture d’écran, Police

Description générée automatiquement**Note** : Ajout du flag **--parents** pour voir les hash des commits parents (ou bien **–-all**)

 **Les barre verticale (|)** montre la progression linéaire des commits dans une branche.

 **Les barres obliques (/)** indiquent la fusion de deux branches, ce qui est représenté par le commit de fusion (89629a9).

 **Les astérisques (\*)** indiquent des commits, avec les hash qui identifient chaque commit.

### Explications des hash avec des parents

#### Commit de fusion (Merge Commit)

* **a6bf079 b864b2f 7fc1887** : Cette ligne représente un commit de fusion, identifié par l'ID de commit **a6bf079**. Les deux autres hash (**b864b2f** et **7fc1887**) sont les commits parents de cette fusion. Cela signifie que le commit de fusion a deux parents, l'un venant de la branche principale (main), et l'autre de la branche fusionnée (add\_classics).
* **(HEAD -> main)** indique que vous êtes actuellement sur la branche main et que ce commit est le plus récent sur cette branche.

#### Commits de la branche add\_classics

* **7fc1887** **5526a09** : Ce commit appartient à la branche add\_classics, et **7fc1887** est l'identifiant de ce commit. Il indique l'ajout de fonctionnalités liées à "add classics". **5526a09** est le parent.

#### Commits de la branche main :

* **b864b2f 5526a09** : Ce commit appartient à la branche main, et **b864b2f** est l'identifiant de ce commit. Il indique l'ajout de fonctionnalités liées à "main ". **5526a09** est le parent.

## Fast forward merge / commit

Une image contenant texte, Police, blanc, conception

Description générée automatiquementUn **merge fast-forward** est la méthode la plus simple de fusion entre deux branches Git. Cela se produit lorsque la **branche de destination contient tous les commits de la branche source**. Git peut alors avancer le pointeur de la branche de destination directement à la pointe de la branche source, **sans créer de commit de fusion.**

**Exemple** : Etat initial :

Une image contenant texte, capture d’écran, Police, Graphique

Description générée automatiquement

Apres merge :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Note** : Le merge fast-forward est souvent utilisé pour fusionner des branches de courte durée ou des modifications mineures, car il simplifie l'historique Git en évitant les commits de fusion inutiles.

# Rebase

## Visualisation d’un rebase

**Le rebase ne crée pas de merge commit**

**La branche qui est rebase ne change pas ! Les commits sont ramenés à l’autre branche**

Une image contenant texte, Police, blanc, diagramme

Description générée automatiquement**Situation initiale :**

Nous avons deux branches avec l'historique suivant :

* **main** : contient les commits partagés par toute l'équipe.
* **feature\_branch** : contient des modifications spécifiques en cours de développement

**Objectif :** Mettre à jour feature\_branch avec les derniers changements de main tout en évitant la création d'un commit de merge.

**Après un Rebase** :

Une image contenant texte, Police, capture d’écran, diagramme

Description générée automatiquementLe rebase **rejoue** les commits D et E de feature\_branch sur la base de la branche main. Cela donne un historique propre et linéaire, comme si D et E avaient été créés directement après C.

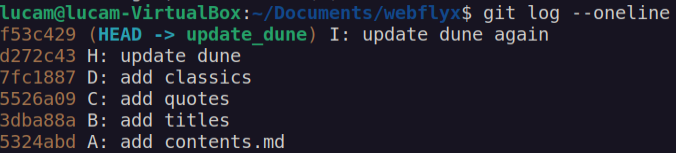
## Exécuter un rebase

### Se positionner dans la branche dans laquelle on veut ramener les commits

**git switch** <BRANCH>

### Commande Rebase

**git rebase** <main | OTHER\_BRANCH>

****Ici nous voyons qu’il manque les commits de main E, F et G dans la branche update\_dune

Une image contenant texte, capture d’écran, Police

Description générée automatiquementAprès le rebase, les commits qui n’étaient pas présent sont ramenés dans la branche update\_dune

## Merge vs Rebase

### Merge

* Conserve **l'historique complet** du projet, montrant quand et où les branches ont été fusionnées.
* Utile pour garder une trace précise du travail collaboratif.
* **Inconvénient** : peut encombrer l'historique avec de nombreux commits de merge, le rendant plus difficile à suivre.

### Rebase :

* Crée un **historique linéaire** en rejouant les commits d'une branche sur une autre.
* Plus facile à lire et à comprendre, surtout pour un travail individuel ou isolé.
* Idéal pour garder les branches de fonctionnalités (features) à jour sans ajouter de commits de merge inutiles.

## Quand utiliser rebase ?

**Branches publiques (par ex. main) :**

* Ne jamais faire un rebase sur une branche publique. Réécrire l'historique d'une branche partagée peut causer des problèmes sérieux, comme casser l'historique pour les autres développeurs.
* Préférez **merge** pour intégrer les modifications sur les branches publiques.

## Quand éviter rebase ?

 **Branches privées (vos branches de fonctionnalités) :**

* Faites régulièrement un rebase pour intégrer les changements de main ou d'autres branches sans polluer l'historique.
* Permet de garder votre branche à jour tout en maintenant un historique propre.

 **Nettoyage avant de partager :**

* Utilisez rebase pour nettoyer ou combiner des commits en unités significatives avant de les partager avec votre équipe.

# Annulation et restauration

## Reset Command

### Soft reset

Cette option de reset revient à un commit précédent tout en gardant toutes les modifications intactes dans l'index (staged).

**git reset** --soft <COMMIT\_HASH>

**Effets :**

* Les changements des commits annulés restent **staged**.
* ****Les changements non commités restent dans leur état initial (staged ou unstaged).

****

### Hard reset

Cette option de reset permet de revenir à un commit précédent tout en supprimant toutes les modifications effectuées après ce commit. Contrairement à --soft, cette option **efface complètement** les changements dans l'index (staged) et dans le répertoire de travail (worktree).

**git reset** --hard <COMMIT\_HASH>

**Effets** :

* **Le HEAD** est déplacé vers le commit spécifié.
* **L'index** et le **worktree** sont mis à jour pour correspondre exactement à l'état du commit.
* ⚠️ Toutes les modifications **non commitées** ou **staged** sont perdues.

## 

### Dangers du hard reset

⚠️ **Attention :** La commande git reset --hard est extrêmement puissante, mais elle peut entraîner des pertes irrémédiables de données. Contrairement à la suppression simple d'un fichier suivi par Git (qui peut être facilement restauré car il est enregistré dans l'historique), **un fichier supprimé via git reset --hard ne pourra pas être récupéré directement**.

**Assurez-vous d’avoir sauvegardé vos changements ou utilisé git stash.**

## Restore command

### Restaurer des fichiers du dernier commit dans le répertoire local

**git restore** <FILE | .>

**Note** : Cela permet de revenir à l'état du fichier tel qu'il était dans le dernier commit, annulant ainsi les modifications locales.

### Annuler un ajout au stage (unstage) :

**git restore** --staged <FILE>

**Note** : Cette commande permet de retirer un fichier de l'index (staged), le rendant à nouveau modifié mais non encore prêt pour le commit.

## Revert Command

**Revert** crée un **nouveau commit** qui **annule exactement les modifications apportées par un commit précédent**. Contrairement à reset, il **ne supprime aucun commit**, ce qui le rend idéal pour des situations où l’historique doit être conservé intact

**git revert** <COMMIT\_HASH>

**Note** : Pour annuler un revert, il faut "**revert le revert**" en utilisant le nouveau commit hash associé au revert

## Récapitulatif

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Commande** | **Historique modifié ?** | **Modifications locales conservées ?** | |  | | --- | |  |  |  | | --- | | **Utilisation typique** | |
| reset | Oui | Non (avec --hard)  Oui (--soft) | Revenir à un état précédent sans garder l'historique. |
| restore | Non | Oui | Annuler des changements locaux ou unstager un fichier. |
| revert | non | Oui | |  | | --- | |  |  |  | | --- | | Annuler un commit tout en conservant l'historique. | |

# Remote

## Gérer des dépôts distants

Les dépôts distants permettent de collaborer avec d'autres développeurs en partageant le code via un référentiel externe. Ces dépôts, appelés **remotes**, conservent généralement un historique Git similaire à votre dépôt local.

Par convention le dépôt distant est appelé origin

## Ajouter un remote

**git remote add** <REMOTE> <REMOTE\_URL>

**Note** : L’URL peut aussi être le chemin relatif du dépôt local dans la machine

**Exemple** : git remote add origin main ajoute main comme dépôt distant "origin"

## Fetch

### Fetch command

La commande git fetch permet de récupérer les modifications d'un dépôt distant (dossier **.git/**) sans les intégrer automatiquement dans vos branches locales.

**git fetch** [<REMOTE>]

**Exemple** : git fetch origin

### Fetch n’intègre pas les commits du dépôt distant

Même après un fetch, les commits ne sont pas encore accessibles dans votre historique local tant que vous n’avez pas explicitement lié votre branche locale à une branche distante ou fusionné les modifications.

### **Conclusion**

Tant que vous n'avez pas explicitement basculé sur une branche distante ou fusionné les commits récupérés, votre historique local restera vide. Cette étape démontre que git fetch **ne modifie pas automatiquement vos branches locales**, ce qui vous donne un contrôle total sur le processus d'intégration

## Log remote

### Afficher l'Historique d'une Branche Distante avec git log

La commande git log peut également être utilisée pour examiner les commits d’une branche distante sans les avoir intégrés à votre branche locale. Cela permet d’avoir un aperçu des changements dans le dépôt distant.

**git log** remote/branch

## Merge Remote

### Fusionner une Branche Distante dans une Branche Locale

Lorsque vous fusionnez une branche distante (origin/main) dans une branche locale (main), cela permet de **synchroniser le contenu du dépôt distant dans votre dossier local**. Cela inclut les nouveaux commits et fichiers.

### Merge remote command

**git merge** remote/branch

**Note** : On fait cette commande dans le **dépôt local** au sein de la **branche main**

# GitHub

## GitHub repository

### GitHub

**GitHub** est la plateforme la plus populaire pour héberger des dépôts Git en ligne. Elle offre plusieurs avantages :

* **Sauvegarde de ton code** : Stocke tes projets dans le cloud, assurant une copie de sauvegarde en cas de problème avec ta machine locale.
* **Collaboration** : Partage ton code et collabore avec d'autres développeurs.
* **Portfolio public** : Utilise GitHub comme vitrine pour tes projets et démontre ton travail.

**Git ≠ GitHub**

Il est essentiel de comprendre que **Git** et **GitHub** ne sont pas la même chose :

* **Git** : Un outil open-source de gestion de versions que tu utilises localement sur ton ordinateur pour suivre les changements apportés aux fichiers de code.
* **GitHub** : Un service web commercial qui héberge des dépôts gérés par Git. Il existe également d'autres plateformes similaires comme **GitLab** et **Bitbucket**.

### GitHub CLI

**GitHub CLI** (Command Line Interface) est un outil qui permet d'interagir avec GitHub directement depuis le terminal. Il facilite les actions courantes comme la gestion des dépôts, l’authentification, l'ouverture de pull requests, la création de nouveaux issues ou la consultation des activités, sans avoir à passer par l'interface web. Cela permet aux développeurs de gagner du temps et d'automatiser certaines tâches liées à GitHub.

## Gestion du dépôt sur GitHub

### Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia Description générée automatiquementUne image contenant texte, capture d’écran, logiciel, Logiciel multimédia Description générée automatiquementCréer un dépôt sur GitHub

### Une image contenant texte, capture d’écran, Police, nombre Description générée automatiquement S’authentifier avec GitHub CLI (gh)

**Une image contenant texte, capture d’écran, Police

Description générée automatiquementgh auth** login / logout

## Remote add command

### Lier le dépôt local au dépot distant (remote) sur GitHub

**git remote add** origin <REPOSITORY\_URL>

### Vérifier la liaison

**git ls-remote**

## Git Push

La commande git push envoie les modifications locales vers un "dépôt distant", comme GitHub. Par exemple, pour pousser les commits de notre branche main locale vers la branche main du dépôt distant, on utiliserait :

**git push** [-u] [--force] origin main

**Note** : **Pour le premier push il faut mettre -u pour lier les dépôts locaux et distants**.

L’option **--force** permet de forcer git à push sur le repository en cas d’erreurs

Il est important d'être authentifié avec le dépôt distant pour pouvoir pousser des modifications

### Options alternatives

Vous pouvez également pousser une branche locale vers une autre branche que main

**git push** origin <BRANCH>

De plus, vous pouvez supprimer une branche distante en poussant une branche vide :

**git push** origin :<REMOTEBRANCH>

## Git Pull

Le git pull permet de récupérer les modifications réelles des fichiers depuis un dépôt distant, pas seulement les métadonnées comme avec git fetch

**git pull** [<REMOTE>/<BRANCH>]

**Note** : Si vous exécutez git pull sans rien spécifier, il récupérera la branche actuelle depuis le dépôt distant par défaut

## Pull request

### Fonctionnement d’un pull request

Le Push effectué pour faire un Pull request doit se faire sur une autre branche que main

Une Pull Request (PR) sur GitHub est un moyen de proposer des changements dans un projet

1. **Collaboration et révision** :
   * Elle permet à d'autres membres de l'équipe ou aux responsables du projet de voir les changements proposés avant qu'ils ne soient intégrés au projet principal..
2. **Flux de travail avec GitHub** :
   * Une fois qu'une PR est ouverte, elle peut être commentée, validée, et même modifiée avant d'être fusionnée dans la branche cible.
3. **Fusion (Merge)** :
   * Après les discussions et les ajustements nécessaires, une PR peut être fusionnée dans la branche cible (souvent main), ce qui intègre les changements dans le code de base.

### Exemple de pull request

#### Partie "Envoyer les modifications"

 Création d'une nouvelle branche à partir de main :

git switch -c <NEW\_BRANCH>

**** Modifications des fichiers

**** Commit les modifications

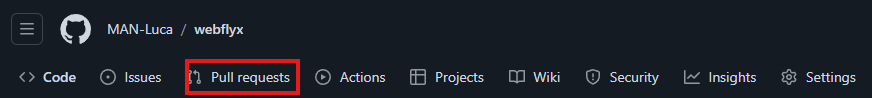
git add .

git commit -m "MESSAGE"

 Push de la nouvelle branche sur GitHub :

git push origin <NEW\_BRANCH>

#### Partie "Création d'une Pull Request"

* Clique sur l'onglet "Pull requests" en haut de la page
* Clique sur "New pull request"

Une image contenant texte, logiciel, Logiciel multimédia, capture d’écran

Description générée automatiquement

* Dans l'interface de création de la PR, choisis **main** comme branche de base et la **nouvelle\_branche** comme branche de comparaison. Puis crée la pull request

Une image contenant texte, logiciel, Police, capture d’écran

Description générée automatiquement

* Pour finir l’administrateur peut décider de merge ou non les modifications

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

# GitIgnore

## Fichier .gitignore

Le fichier .gitignore permet de spécifier quels fichiers ou répertoires Git doit ignorer lors des opérations de suivi de fichiers dans un dépôt.

### Exemple d’utilisation

Exemple, si tu écris dans un fichier .gitignore :

node\_modules

Git ignorera **tous** les répertoires ou fichiers appelés node\_modules à n'importe quel niveau du répertoire. Cela inclut :

* node\_modules/code.js
* src/node\_modules/code.js
* src/node\_modules/

Cependant, cela **ne** va **pas** ignorer des chemins comme :

* src/node\_modules\_2/code.js
* env/node\_modules\_3

Cela montre que Git va uniquement ignorer les fichiers ou dossiers exactement nommés node\_modules et leur contenu, sans toucher aux autres chemins similaires.

## Nested .gitignore

Dans un projet, il est possible d'avoir plusieurs fichiers .gitignore placés dans différents répertoires. Ces fichiers **fonctionnent de manière hiérarchique et ne s'appliquent qu'au répertoire où ils se trouvent et à ses sous-répertoires.**

### Fonctionnement

 **Fichier** .gitignore **dans le répertoire racine (unique):**

* Les règles définies dans ce fichier s'appliquent à l'ensemble du projet, sauf si elles sont contredites par un fichier .gitignore dans un sous-répertoire.

 **Fichiers** .gitignore **dans des sous-répertoires (peuvent être multiples) :**

* Ces fichiers remplacent ou ajoutent des règles spécifiques pour le répertoire où ils se trouvent et ses sous-répertoires.

### Exemple concret

Arborescence :

/project

.gitignore

/src

.gitignore

/components

file.js

temp.log

build/

/docs

temp.doc

Contenu du .gitignore à la racine :

\*.log

build/

Contenu du .gitignore dans /src :

!build/

\*.tmp

**Résultat** :

1. **À la racine** :
   * Les fichiers .log sont ignorés partout dans le projet.
   * Le dossier /src/build est suivi car le .gitignore imbriqué utilise !build/ pour annuler l'exclusion.
2. **Dans /src :**
   * Les fichiers .tmp dans /src et ses sous-dossiers sont ignorés.
   * Les fichiers .log restent ignorés à cause de la règle racine.
3. **Dans /docs :**
   * Le fichier temp.doc n'est pas affecté par le .gitignore dans /src.

## Patterns

### Wildcard (\*)

**Le caractère \* remplace n’importe quel caractère sauf /**

**Exemple** : \*.txt Ignore tous les fichiers avec l'extension .txt

### Double wildcard (\*\*)

**\*\*/foo : Correspond à "foo" partout dans l'arborescence.**

**/foo/\*\* : Correspond à tout contenu dans "foo" avec une profondeur infinie.**

### Rooted patterns (/)

Les patterns commençant par / s’appliquent uniquement au répertoire contenant le fichier .gitignore

**Exemple** : /main. Ignore uniquement le fichier main.py situé dans le répertoire racine, mais pas dans les sous-dossiers.

### Négation

**Le caractère ! annule une règle précédente pour inclure des fichiers ou dossiers spécifiques**

**Exemple** :

\*.txt

!important.txt

Ignore tous les fichiers .txt sauf important.txt

### Commentaires

**Toute ligne commençant par # est un commentaire, utile pour documenter les règles**

**Exemple** :

# Ignorer tous les fichiers logs

\*.log

## Que mettre dans le .gitignore ?

### Fichiers générés

Il s'agit des fichiers qui sont créés automatiquement par des processus de compilation ou de construction. Ils peuvent être régénérés à partir du code source ou de la configuration, il n'est donc pas nécessaire de les suivre avec Git.

* **Code compilé :** Fichiers comme .class, .o, .exe, .dll, .pyc, .pyo, etc.
* **Fichiers minifiés :** Fichiers générés pour la production comme .min.js, .min.css.
* **Fichiers de logs et temporaires :** Fichiers comme .log, .bak, .swp ou ceux créés lors des tests.

### Dépendances

Les dépendances sont gérées par des gestionnaires de paquets, il est donc préférable de ne pas les suivre. Si vous les suivez, votre dépôt va grossir inutilement et les changements de dépendances devraient être gérés par le gestionnaire de paquets.

* **Node modules (node\_modules)** pour les projets JavaScript.
* **Environnements virtuels (venv, env)** pour les projets Python.
* **Répertoires de paquets** comme vendor/ pour PHP, libs/ pour Java, ou .bundle/ pour Ruby.
* **Fichiers spécifiques au système** pour différents environnements comme \*.dmg, .app, ou .vs pour Visual Studio.

### Fichiers personnels ou spécifiques à l'éditeur

De nombreux développeurs utilisent des configurations personnalisées pour leurs éditeurs, IDE ou autres outils. Ces fichiers sont spécifiques à l'environnement de travail local et ne sont pas nécessaires pour le projet ou l’équipe.

* **Paramètres d'éditeur :** Fichiers comme .vscode/, .idea/, .editorconfig, .sublime-project.
* **Fichiers spécifiques au système d'exploitation :** Fichiers comme .DS\_Store (macOS), Thumbs.db (Windows), ou desktop.ini.

### Informations sensibles ou dangereuses

Il est crucial d'éviter de suivre des fichiers contenant des informations sensibles, car ils pourraient exposer des mots de passe, des clés API et d'autres secrets qui doivent rester privés. Ces fichiers ne doivent jamais être dans un dépôt public ou partagé.

* **Fichiers d'environnement** comme .env, .env.local, qui peuvent contenir des clés API, des secrets ou des informations de connexion à une base de données.
* **Clés privées ou certificats** comme .pem, .key, .crt.
* **Fichiers de base de données** comme \*.sqlite, \*.db qui peuvent contenir des données personnelles, financières ou autrement sensibles.

# README.md

## Qu’est-ce QU’UN fichier README ?

Un fichier **README.md** est essentiel dans un projet / repository pour expliquer son objectif, son utilisation et ses détails techniques. Il sert de **première impression** pour les collaborateurs, utilisateurs ou recruteurs.

Ce fichier doit être **placé à la racine du dépôt local** et doit s’appeler **README.md** (md signifie **markdown**)

GitHub reconnait automatiquement ce fichier et l’affiche sur la page d’accueil du repository

## Syntaxe d’un fichier .md

### Titres

Utilisez des signes dièse (#) pour créer des titres. Plus il y a de dièses, plus le titre est de niveau bas.

**Exemple** :

# Titre de niveau 1

## Titre de niveau 2

### Titre de niveau 3

### Gras et italique

 **Gras** : Entourez le texte avec deux astérisques \*\* ou deux underscores \_\_.

 **Italique** : Entourez le texte avec un astérisque \* ou un underscore \_.

\*\*Texte en gras\*\*

\*Texte en italique\*

### Listes

 **Listes à puces** : Utilisez des astérisques \*, des tirets -, ou des plus +.

 **Listes numérotées** : Utilisez des numéros suivis d'un point.

- Élément 1

- Élément 2

- Sous-élément 1

- Sous-élément 2

\* Élément 3

1. Première étape

2. Deuxième étape

### Liens et images

Pour ajouter un lien, utilisez cette syntaxe. Les images sont ajoutées de manière similaire aux liens, mais avec un point d'exclamation au début :

<!>[Texte](URL)

### Bloc de code avec coloration syntaxique

* **Code en ligne** : Entourez le texte de backticks (`).
* **Blocs de code** : Utilisez trois backticks (```) avant et après le code.

```<LANGUAGE>

Lignes de codes

```

### Tableaux

Les tableaux sont créés en utilisant des barres verticales | et des tirets - pour délimiter les colonnes et les lignes.

| Titre 1 | Titre 2 |

|---------|---------|

| Ligne 1 | Donnée 1 |

| Ligne 2 | Donnée 2 |

### Saut de ligne

Pour forcer un saut de ligne, vous pouvez ajouter deux espaces à la fin de la ligne avant de presser "Entrée".

### Citations

> Ceci est une citation.

### Séquences d’échappement

Si vous avez besoin d’afficher des caractères spéciaux (comme des astérisques, des dièses, etc.), utilisez un backslash \ devant ces caractères.

\\*Texte entre astérisques\\*

## Exemple complet de fichier .md

### Code markdown

# Mon Projet

## Description

\*\*Mon projet\*\* est un projet open source que j'ai créé pour apprendre la programmation en Java.

### Fonctionnalités

- Fonction 1

- Fonction 2

- Fonction 3

## Installation

1. Clonez le dépôt :

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

2. Allez dans le répertoire :

`cd projet`

## Contribution

1. Forkez le projet.

2. Créez une nouvelle branche pour votre fonctionnalité.

3. Faites une Pull Request.

## Liens utiles

- [GitHub](https://github.com)

- [Mon blog](https://monblog.com)

> "La simplicité est la sophistication suprême." – Léonard de Vinci

### Code Extrait

Voici un exemple de code simple en Java :

```java

public class Main {

public static void main(String[] args) {

System.out.println("Bonjour, Monde!");

}

}

```

### Une image contenant texte, capture d’écran, logiciel Description générée automatiquementAffichage sur GitHub

## Fichier readme de profil GitHub

### Marche à suivre

Pour que le fichier **README** apparaisse sur votre profil GitHub, vous devez **créer un dépôt avec le même nom que votre nom d'utilisateur GitHub**. (Laissez le dépôt en **public**) Ensuite, mettre un fichier README.md dans le dépôt. Maintenant vous avez une présentation sur votre profil GitHub

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

### Exemple de présentation de profil GitHub

# Bonjour, je suis Luca Mansutti 👋

Je suis un développeur passionné par la programmation Java, l'utilisation de Git/GitHub et la programmation en assembleur pour microcontrôleurs.

## 🛠️ Compétences

- \*\*Langages\*\* : Java, Python, Assembleur (PIC16F84)

- \*\*Outils\*\* : Git, GitHub, Eclipse, MPLABX

- \*\*Autres\*\* : Documentation technique, Gestion de projet

## 🚀 Projets

- [Pong en Java](https://github.com/username/pong) : Jeu rétro avec des fonctionnalités avancées.

- [Carnet de contacts](https://github.com/username/carnet) : Application de gestion de contacts en Java.

## 📫 Me contacter

- \*\*Email\*\* : luca.mansutti@example.com

- \*\*LinkedIn\*\* : [Luca Mansutti](https://linkedin.com/in/luca-mansutti)

Une image contenant conception

Description générée automatiquement avec une confiance faible

Partie 2 : Concepts avancés

Une image contenant symbole, clipart, créativité

Description générée automatiquement

# fork

## Qu’est-ce qu’un fork ?

Un fork est une **copie d’un dépôt Git**, permettant de modifier un projet sans impacter l’original et de proposer des changements via des pull requests. "Forker" un repository permet donc de ramener une copie de celui-ci au sein de **vos propres repositories**

## Une image contenant texte, capture d’écran, Police, Logiciel multimédia Description générée automatiquementComment fork un repository

Ensuite, le repository est copié dans vos repositories personnels

## Exemple réel de travail collaboratif

### Cloner le dépôt "forké"

**git clone** <REPOSITORY\_URL>

### Créer une nouvelle branche pour les nouvelles fonctionnalités

**git switch** -c <BRANCH>

### Ajouter et commit les modifications

**git add** .

**git commit** -m "MESSAGE"

### Pousser les changements vers GitHub

**git push** -u origin <BRANCH>

**Note** : Il est conseillé de push vers une branche autre que main

### Créer un pull request

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquementCe pull request se fait sur **votre repository forké** (B), mais les changements devront être amenés sur **le repository de base** (A)

# Reflog

## Head

**HEAD** représente **l'endroit où vous vous trouvez actuellement** dans votre dépôt. Plus précisément, HEAD est un pointeur qui indique :

* **La branche active** sur laquelle vous travaillez.
* **Le dernier commit** référencé par cette branche.

### Voir où pointe HEAD

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement**cat** .git/HEAD

## Reflog command

**git reflog**

### Explications de reflog

La commande **git reflog** (abréviation de **reference log**) est un outil qui enregistre l'historique des changements effectués sur une référence (comme HEAD ou une branche). Contrairement à **git log**, qui affiche les commits dans l'ordre chronologique, **git reflog** se concentre sur **les déplacements récents de HEAD** ou d'autres références.

### Fonctionnement de reflog

Chaque fois que HEAD (ou une branche) change d'état ou de position (par exemple, lors d'un commit, d'un merge, ou d'un reset), Git l'enregistre dans le reflog. Cela vous permet de retrouver des commits perdus ou de suivre les actions effectuées dans votre dépôt

|  |  |
| --- | --- |
| HEAD@ {0} | Position actuelle de HEAD |
| HEAD@ {0} | Position de HEAD 1 action en arrière |
| HEAD@ {2} | Position de HEAD 2 action en arrière |
| HEAD@ {3} | Position de HEAD 3 action en arrière |

### Log VS Reflog

|  |  |
| --- | --- |
| git log | git relfog |
| Historique des commits | Historique des mouvements de HEAD |
| Affiche les commits visibles | Inclut même les commits "perdus" ou cachés |
| Format axé sur les hashes et les branches | Format basé sur HEAD@{} pour retracer les étapes |

## Récupération du contenu d’un fichier supprimé

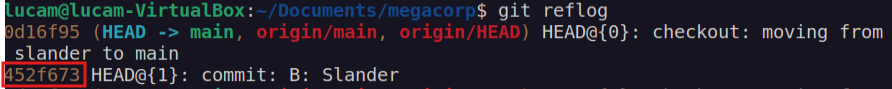
**git reflog** est un outil puissant pour retracer les actions dans un dépôt Git. Il est particulièrement utile pour **récupérer des commits perdus** ou **comprendre les modifications de HEAD**.

### Étapes à suivre pour récupérer le contenu d’un fichier supprimé

#### Vérifier l’historique avec reflog

**git reflog**

Permet de retrouver le **COMMIT\_HASH**



#### Explorer le commit avec git cat-file -p

**git cat-file** -p <COMMIT\_HASH>

Une fois que vous avez le hash du commit, inspectez son contenu pour retrouver le **tree\_hash**

**Une image contenant texte, capture d’écran, Police, information

Description générée automatiquement**

#### Explorer le tree

**git cat-file** -p <TREE\_HASH>

Une fois que vous avez le hash du tree inspectez son contenu pour retrouver le **blob\_hash**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

#### Examiner le blob pour retrouver le contenu du fichier

**git cat-file** -p <BLOB\_HASH>

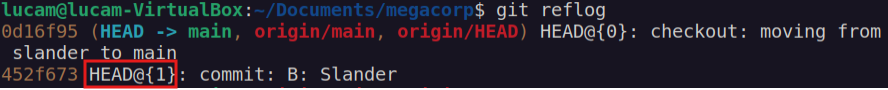
## Récupération améliorée avec merge

La méthode manuelle avec cat-file est fastidieuse. Heureusement nous pouvons remplacer toutes ces étapes par une seule commande

### Merge avec un commitish

**git merge** <COMMITISH>

**Note** : commitish = HEAD@ {n}

****



# Merge conflicts

## Introduction aux conflits

Les conflits se produisent lorsque plusieurs développeurs modifient les **mêmes lignes** dans un fichier ou lorsque des modifications contradictoires sont apportées sur des branches différentes **sans relation parent-enfant**. Cela arrive souvent lors des **fusions** ou des **rebases**.

### Exemple de conflit

Une image contenant texte, Police, blanc, conception

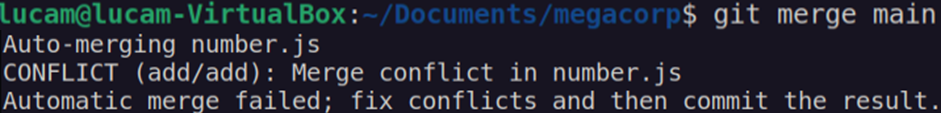
Description générée automatiquement

* Une image contenant texte, capture d’écran, Police, ligne

  Description générée automatiquementFichier "number.js" dans la branche main (commit B) :
* Une image contenant texte, capture d’écran, Police, ligne

  Description générée automatiquementFichier "number.js" dans la branche feature (commit C) :

Si vous tentez de fusionner "feature" dans "main", Git détecte que la **même ligne** a été modifiée dans les deux branches **sans relation parent-enfant**. Cela crée un **conflit de fusion**.



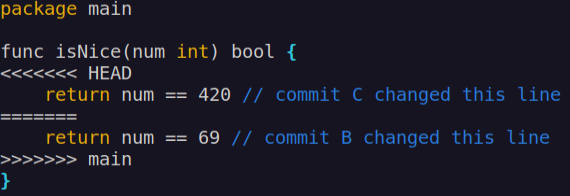
Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## Résoudre un conflit

### Localiser le conflit

Lorsque Git détecte un conflit, il marque le fichier concerné avec des **marqueurs de conflit**



La première section entre <<<<<<< HEAD et ======= représente **la branche dans laquelle vous êtes**.

La seconde section entre ======= et >>>>>>> main représente **la branche avec laquelle vous voulez faire le merge**

### Éditer les fichier et résoudre les conflits

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquementSupprimez les marqueurs et **combinez les deux** modifications, où bien **supprimez une des deux**)

.

### Ajouter et finaliser la fusion

**git add** .

**git commit** -m "E : Résolution des conflits"

**Note** : Lors d’un conflit de merge, git rentre dans un mode spécial. Si les conflits sont résolus, alors le prochain commit clôturera le merge des branches.

### Vérifier le merge

**git log**

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## Arrêter un merge lors d’un conflit

**git merge** --abort

**Note** :

* Cette commande annule la fusion en cours et restaure l'état de votre dépôt comme il était avant le début de la fusion.
* Elle supprime tous les fichiers conflictuels ou partiellement modifiés introduits par la tentative de merge.

## Outils de résolution de conflits intégrée à Git

### "Ours" vs "Theirs" (Notre vs Leurs)

Lorsque vous êtes en conflit de fusion dans Git, les termes **"ours"** et **"theirs"** désignent les branches impliquées dans le conflit :

* **"Ours"** : cela fait référence à la branche sur **laquelle vous êtes** actuellement (la branche dans laquelle vous êtes en train de fusionner).
* **"Theirs"** : cela fait référence à la branche que **vous fusionnez dans votre branche actuelle.**

### Checkout merge command

**git checkout** –-theirs/--ours <PATH>

**Note** :

* **--ours** va **écraser** le fichier avec les modifications **de la branche sur laquelle vous êtes** actuellement et vers laquelle vous fusionnez.
* **--theirs** va écraser le fichier avec les modifications **de la branche que vous êtes en train de** fusionner avec la branche courante.

Une fois les problèmes résolus nous pouvons faire :

**git add** .

**git commit** -m "MESSAGE"

# Rebase conflicts

## Exemple de rebase conflicts

### Scénario :

* Vous travaillez sur une branche banned et souhaitez la mettre à jour avec les dernières modifications de main. Vous utilisez un *rebase*. Pendant ce temps, d'autres ont ajouté des modifications sur main

Premier fichier banned.csv de la branche "banned" contenant :

first\_name,last\_name,company,title

Kayha,tbd,TheMarchOfTime,sidekick

sam,ctrlman,closedai,ceo

Second fichier banned.csv de la branche "main" contenant :

first\_name,last\_name,company,title

Ballan,Agrandian,Boots.lore,Protagonist

sam,ctrlman,closedai,ceo

* En *rebasant*, vous essayez d'ajouter les changements de banned au-dessus de main, ce qui entraîne un conflit si les mêmes fichiers ont été modifiés dans les deux branches.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Vous devez résoudre ce conflit dans un état spécial de Git appelé ***detached HEAD***.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

### L'état "detached HEAD"

Lors d'un *rebase*, le HEAD montre la branche cible (main dans cet exemple), et non la branche source (banned). Vous êtes dans un état de "**detached HEAD**", vous permettant de résoudre les conflits avant de finaliser le processus de rebase. Cela signifie que vous n'êtes plus sur une branche spécifique mais sur un commit temporaire Résoudre les rebase conflicts

### Checkout rebase command

**git checkout** –-theirs/--ours <PATH>

**Attention** :

* **--ours** va **conserver** les modifications de la **branche en cours de rebase**
* **--theirs** va remplacer le contenu du fichier par les **modifications de la branche cible (main)**

### Terminer le rebase

**git rebase** --continue

**Note** : Contrairement aux merge conflicts qui nécessitent de commit pour terminer les modifications, le rebase nécessite –continue

# RERERE (Reuse recorded resolution)

## Qu’est-ce que git rerere

Dans Git, les conflits répétitifs peuvent être frustrants, surtout lorsque vous travaillez sur des branches de fonctionnalités qui nécessitent un **rebase** ou un **merge** fréquent avec main. Heureusement, Git propose une solution appelée rerere, qui signifie **reuse recorded resolution** :

* rerere enregistre la manière dont vous avez résolu un conflit spécifique.
* Si Git rencontre le **même conflit** à nouveau, il applique automatiquement la résolution que vous aviez utilisée précédemment.

Cela fonctionne avec les **rebases** et les **merges**

### Activer / désactiver rerere

**git config** [--local | --global] rerere.enabled {TRUE|FALSE}

### Supprimer le dossier rerere

**rm** -rf .git/rr-cache

## Exemple d’utilisation de rerere

Non avons 3 branches : **main**, **favs** et **favs2**

Le but est de résoudre les conflits manuellement lors du rebase de main sur favs, ensuite de rebase main sur favs2. Pour le second rebase rerere permettra de ne pas devoir faire les changements manuellement, car il aura enregistré la manière de résoudre ce type de conflits.

### Premier rebase de main sur favs (Entrée dans la zone de conflit)

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Note** : La ligne encadrée montre que Git enregistre la manière de gérer ce type de conflit.

### Résolution des conflits et fermeture du rebase

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Note** : La ligne encadrée montre que Git a enregistré la manière de gérer ce type de conflit.

### Rebase de main sur favs2 (Résolution automatique)

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Note** : La ligne encadrée montre que Git à résolu tout seul le conflit. Nous n’avons donc pas du gérer les conflits manuellement

### Fermeture du rebase

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## ****Annuler un commit (alors qu’il fallait --continue)****

### Problème courant

Lors de la résolution d'un conflit pendant un **rebase**, il est facile d'oublier que l'on doit utiliser git rebase --continue au lieu de créer un commit. Si vous "commitez" accidentellement la résolution, Git enregistre ce commit dans l'historique du rebase, ce qui peut causer des incohérences.

### Annuler un commit accidentel

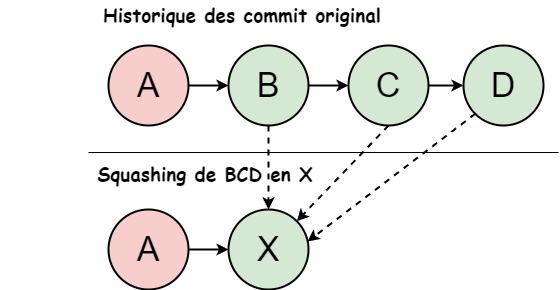
**git reset** --soft HEAD~1

**Note** : L'option --soft préserve vos modifications dans le staging area, sans supprimer les résolutions de conflit.

# Squash

## Qu’est-ce que le squashing ?

Le squashing consiste à combiner plusieurs commits en un seul. Cela permet de simplifier l’historique Git et de le rendre plus lisible



**Attention** : Le squashing est **destructif**, il faut donc toujours le faire sur une **branche temporaire** (temp\_main), puis remplacer main par la nouvelle branche temp\_main et pour finir renommer temp\_main en main

## Comment faire un squash ?

### Démarrer un rebase interactif

**git rebase** -i HEAD~n

**Note** : Remplacez n par le nombre de commits que vous voulez inclure dans le squash.

Pour **inclure le commit initial** : git rebase -i --root

### Choisir les commits à squash

* Une fois la commande exécutée, une interface interactive s'ouvre dans votre éditeur par défaut. Chaque ligne correspond à un commit récent, comme ci-dessous :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Changez **pick** en **squash** (ou simplement s) pour tous les commits que vous voulez fusionner avec le premier. Par exemple :

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* Après avoir sauvegardé, Git vous demandera de rédiger le message pour le nouveau commit unique (supprimer les messages des anciens commits que vous voulez squash)

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* Une image contenant texte, capture d’écran, Police

  Description générée automatiquementLe squashing à bien été effectué

## Overwrite main

Maintenant que nous avons fini d’actualiser la branche temporaire (temp\_main), nous pouvons remplacer la branche main par celle-ci

**Note** : Cette méthode n’est pas recommandée. La méthode classique est de créer une pull-request **(voir 16.6**)

### Supprimer la branche main

**git branch** -D main

### Renommer la branche temporaire par main

**git branch** -m main

## Dangers du squashing

Squasher les commits peut être intimidant, car cela efface l'historique des modifications. Par exemple, si vous avez une série de commits comme :A - B - C - D

et que vous les réduisez en un seul commit : ABCD

vous supprimez les repères de chaque modification individuelle. Bien que toutes les modifications soient toujours présentes dans l’historique final, les points de contrôle individuels sont supprimés. Cela signifie que, une fois le squash effectué, vous ne pourrez plus revenir aux commits précédents pour examiner l’évolution du projet à chaque étape.

## Squashing pull requests

### Résumé du flux de travail

1. Créez une nouvelle branche à partir de main.
2. Travaillez sur la branche de fonctionnalité en ajoutant des commits au fur et à mesure.
3. Une fois prêt, combinez tous vos commits en un seul.
4. Poussez la branche vers le dépôt distant.
5. Ouvrez une pull request depuis votre branche de fonctionnalité vers main.
6. Une fois la pull request approuvée, effectuez la fusion.

## Force push

### Qu’est-ce qu’un force push

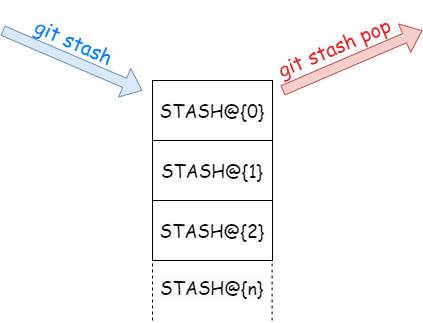
Le force push nécessaire lorsque l’historique de votre branche locale est incompatible avec celui de la branche distante. Cela peut arriver après des opérations comme un **rebase** ou un **squash**, qui modifient l’ordre ou le contenu des commits. Dans ces situations, Git bloque un push classique afin de protéger l’historique de la branche distante.

### Force push command

**git push** origin main --force

# Stash

## A quoi sert le stashing ?

La commande git stash **enregistre l'état actuel** de votre répertoire de travail et de l'index (zone de staging). Cela fonctionne un peu comme un presse-papiers : elle place temporairement les modifications de côté et revient à l'état du dernier commit de votre branche.

## Stash command

### Créer un stash

**git stash** [-m "MESSAGE"]

### Lister les stashes

**git stash** list

### Stash Pop

La commande git stash pop permet de réappliquer les modifications que vous avez "stash" précédemment et de les supprimer de la liste des stashes.

**git stash** pop stash@{n}

**Note** : Permet de réappliquer vos changements à votre répertoire de travail et **supprimer** l'entrée du stash

### Stash Apply

**git stash** apply stash@{n}

**Note** : Permet de réappliquer vos changements à votre répertoire de travail **MAIS ne supprime pas** l'entrée du stash

### Stash drop

**git stash** drop stash@{n}

**Note** : Permet de supprimer une entrée du stash **sans y retourner**

# Diff

## Diff command

La commande git diff vous permet de **voir les différences entre différents états de votre code**..

### Comparer l’état actuel et le dernier commit

**git diff**

### Comparer un commit précédent et l'état actuel

**git diff** HEAD~{n}

### Comparer deux commits spécifiques

**git diff** <COMMIT\_HASH\_1> <COMMIT\_HASH\_2>

### Comparer l'index (zone de staging) avec la dernière validation

**git diff** --cached

### Comparer l'arbre de travail avec l'index

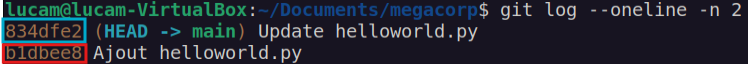
**git diff** HEAD

## Exemple d’utilisation de diff

Une image contenant Police, texte, capture d’écran, Graphique

Description générée automatiquementUne image contenant Police, texte, capture d’écran, Graphique

Description générée automatiquement Commit "Ajout helloworld.py" Commit "Update helloworld.py"

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## Blame command

La commande git blame permet d’identifier l’auteur et l’origine de chaque ligne dans un fichier.

**git blame** [OPTIONS] <FILE>

**Options** :

* -L <start>,<end> : Affiche uniquement les lignes spécifiées (ex. lignes 5 à 15).
* -e : Montre l’adresse email de l’auteur en plus de son nom.
* -C : Suit les changements dans les fichiers copiés.
* -M : Suit les lignes déplacées ou renommées dans un fichier.

### Exemple de sortie de git blame

e94a23fc (John Doe 2023-11-01 12:45:32 +0100) if est\_gentil:

e94a23fc (John Doe 2023-11-01 12:45:32 +0100) print("Merci")

a5d67b2d (Jane Smith 2023-10-29 15:10:12 +0100) return 0

Informations affichées

* **Commit hash** : Identifie le commit responsable de la ligne (les 8 premiers caractères suffisent en général).
* **Auteur** : La personne ayant modifié la ligne.
* **Date** : Quand la ligne a été modifiée.
* **Contenu** : Le texte exact de la ligne

# Cherry pick

## Que fait la commande cherry-pick ?

**Extrait un commit spécifique** d’une branche sans pour autant merge ou rebase toute la branche

 **Rejoue les changements** du commit spécifié sur la branche actuelle.

 Crée un **nouveau commit** avec le même contenu mais un nouvel identifiant, tout en conservant l’historique de la modification.

## Cherry-pick command

**git cherry-pick** <COMMIT\_HASH>

En cas de conflits, vous devrez résoudre manuellement les différences, ajouter les fichiers résolus avec git add, puis continuer avec :

**git cherry-pick** --continue

Si nécessaire, annulez l’opération avec :

**git cherry-pick** --abort

# Bisect

## A quoi sert le bisect ?

git bisect utilise une **recherche dichotomique** pour **identifier un commit spécifique**, ce qui réduit considérablement le nombre de vérifications nécessaires.

**Bisect** est utilisé pour :

* **Bugs** : Identifier le commit qui a introduit un comportement erroné.
* **Régressions** : Trouver le commit à l'origine d'une baisse de performance ou d'un problème.
* **Changements non souhaités** : Localiser la source d'une modification indésirable.

## Marche à suivre du bisect

### Commencer le bisect

**git bisect** start

### Marquer un commit fonctionnel (bon commit)

**git bisect** good <COMMIT\_HASH>

### Marquer un commit problématique (mauvais commit)

**git bisect** bad <COMMIT\_HASH>

### Tester le commit intermédiaire

 Git vous positionne sur un commit entre les points "bon" et "mauvais".

 Testez si le bug est présent.

 Indiquez le résultat (étape suivante)

### Déclarer le résultat du test

**git bisect** good # Le commit est "bon".

**git bisect** bad # Le commit est "mauvais".

### Répéter les 2 étapes précédentes

Git continue d'affiner la recherche en testant les commits restants jusqu'à identifier celui qui a introduit le bug.

### Terminer la session de bisect

Une fois le commit trouvé, quitter le mode bisect :

**git bisect** reset

# Worktree

## Qu’est-ce qu’un worktree ?

Un **worktree** est une arborescence de travail associée à un dépôt Git. C'est un espace où vous pouvez manipuler les fichiers suivis par Git. Chaque dépôt possède :

* Un **main worktree** (principal) : là où se trouve le répertoire .git.
* D'éventuels **linked worktrees** (liés) : des arborescences légères, connectées au dépôt principal.

## Types de worktrees

### Main Worktree

* Contient le répertoire .git avec **toutes les données du dépôt**.
* Consomme plus d'espace disque car il contient tout l'historique Git.
* Nécessite un **git clone** ou un **git init** pour être créé.

### Linked Worktree

* Contient un fichier .git qui pointe vers le main worktree.
* Très léger, comparable à une branche.
* Utile pour travailler sur plusieurs branches ou ensembles de modifications sans perdre de temps à stasher ou changer de branche.

## Worktree commands

### Lister tous les worktrees associés à un repository

**git worktree** list

### Créer un linked worktree

**git worktree** add <PATH> [<BRANCH>]

**Note** : <path> : emplacement du nouveau worktree. <branch> (optionnel) : branche à utiliser. Si omis, utilise le dernier segment du chemin comme nom de branche.

### Afficher le contenu du fichier .git d’un linked worktree

**cat** <PATH> >/.git

## pas de branches partagées entre worktrees !

Les **linked worktrees** se comportent comme des dépôts Git normaux :

* Vous pouvez créer/supprimer des branches, passer d'une branche à une autre, créer des tags, etc.

**Cependant,** une restriction importante s'applique :

**Une branche ne peut être utilisée simultanément par plusieurs worktrees**

**Exemple** : Main worktree (**megacorp**) et linked worktree (**ultracorp**)

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Résultat :** Vous ne pouvez pas utiliser main dans deux worktrees en même temps.

**Solution** : Travailler sur une nouvelle branche

## Tracking des linked worktrees

Le dossier .git/worktrees centralise les informations de tracking pour chaque linked worktree. Cela permet à Git de les gérer efficacement tout en maintenant une connexion au dépôt principal.

**ls** .git/worktrees

**Note** : Le dossier .git/worktrees contient un sous-dossier pour chaque linked worktree.



## Main et linked worktrees : Système unique

Les linked worktrees fonctionnent directement avec le **dépôt principal**, car ils ne possèdent pas de répertoire .git. Cela signifie que toute modification dans un linked worktree est immédiatement reflétée dans le main worktree, et vice-versa.

Lorsque vous exécutez git branch, Git indique si une branche est déjà utilisée dans un autre worktree en affichant un message spécifique à côté de son nom. Cela vous permet de savoir quelle branche est actuellement "occupée" par un autre worktree.

## Supprimer un worktree

### Supprimer un worktree ne supprime pas la branche

La commande git worktree remove permet de supprimer un worktree **sans toucher à la branche associée**. Le worktree est uniquement une vue de votre dépôt, et il est séparé des branches réelles.

### Remove command

**git worktree** remove <WORKTREE-NAME>

### Prune command

Pour nettoyer complètement les références aux worktrees supprimés, vous pouvez utiliser la commande :

**git worktree** prune

**Note** : Cela va supprimer toute référence restante aux worktrees supprimés dans le dossier .git/worktrees de votre dépôt.

# Tags

## Qu’est-ce qu’un tag ?

Un **tag** est un marqueur **immobile** lié à un commit spécifique. Contrairement aux branches, les **tags ne changent pas avec de nouveaux commits**. Ils sont principalement utilisés pour **marquer des versions importantes** ou des jalons dans un projet.

Une image contenant écriture manuscrite, texte, Police, calligraphie

Description générée automatiquement**Fun Fact** : Dans Git, vous pouvez utiliser un **tag** à la place d’un **hash de commit** dans la plupart des commandes, car les tags sont une forme de "**commitish**" (une référence vers un commit)

## Tag commands

### Lister les tags existants

#### Localement

**git tag**

#### Sur un dépôt distant

**git ls-remote** --tags

### Afficher les détails d’un tag

**git show** <TAG\_NAME>

### Créer un tag annoté

**git tag** -a "TAG\_NAME" <COMMIT\_HASH> -m "TAG\_MESSAGE"

**Note** : L’ajoute du **COMMIT\_HASH** permet de **créer un tag sur un commit spécifique**

### Supprimer un tag

#### Localement

**git tag** -d "TAG\_NAME"

#### Sur le dépôt distant

**git push** origin --delete <TAG\_NAME>

### Pousser les tags vers le dépôt distant

**git push** origin --tags

Une image contenant conception

Description générée automatiquement avec une confiance faible

Une image contenant symbole, clipart, créativité

Description générée automatiquementFormulaire des commandes

# Commandes fondamentales

**Repository**

|  |  |
| --- | --- |
| **git init** | Initialise un dossier comme dépôt git |
| **git status** | Affiche l’état des fichiers d’un dépôt |
| **git add** <FILE | .> | Ajoute des fichiers au staging |
| **git rm** --cached <FILE | .> | Retire des fichiers du staging (arrêter de suivre) |
| **git restore** --staged <FILE | .> | Retire des fichiers du staging (continuer de suivre) |
| **git commit** -m "MESSAGE" | Faire un commit avec message |
| **git commit** -am "MESSAGE" | Add + commit en une ligne (si fichier déjà suivi) |
| **git commit** –amend -m "MESSAGE” | Modifier message du dernier commit |

**Log**

|  |  |
| --- | --- |
| **git log** [OPTIONS] | Affiche historique complet des commits et leur hash |

|  |  |
| --- | --- |
| --oneline | Affiche les commits d’une manière simplifiée |
| --graph | Affiche l’arborescence des commits |
| --parents | Affiche les commits hash des parents de chaque commit |
| -n <nombre> | Limite l’affiche à x commits |
| --all | Affiche les commits de TOUTES les branches |
| --stat -p | Affiche un résumé des fichiers modifiés + n° lignes |
| –-decorate=<full | short | no> | Affiche toute/normal/pas de références |
| <remote>/<branch> | Affiche l’historique d’une branche distante |

**Fonctionnement interne**

|  |  |
| --- | --- |
| **ls** -al .git/objects | Affiche contenu du dossier .git/objects |
| **ls** -al .git/objects/<COMMIT\_HASH> | Trouve l’objet correspondant à un commit |
| **git cat-file** -p <HASH> | Affiche le contenu d’un commit/tree/blob |
| **find** .git/refs/heads -type f | Afficher toutes les branches d’un dépôt |
| **cat** .git/refs/heads/<BRANCH> | Afficher le COMMIT\_HASH d’une branche |

**Configurations**

|  |  |
| --- | --- |
| **git config** [options] **<SECTION>.<KEY> "valeur"** | Gérer les configs globales/locales |
| **git config** –-list | Consulter config de Git |

|  |  |
| --- | --- |
| --global | Modifie globalement (fichier .gitconfig) |
| --local | Modifie localement (fichier .git local) |
| --add | Ajoute une configuration |
| --unset | Supprime une configuration |
| --unset –all <section>.<key> | Supprime toutes les occurrences d’une clé |
| –-remove-section <section> | Supprime toute une section |

Config de git :

* **user.username "nom"**  : Set le nom de l’utilisateur
* **user.email "email"**  : Set l’email de l’utilisateur
* **init.defaultBranch "nom\_branche"**  : Set le nom de la branche par défaut
* **core.editor "nom\_éditeur"**  : Set l’éditeur par defaut
* **core.ignorecase "boolean" :** Set la casse par défaut

**Branching**

|  |  |
| --- | --- |
| **git branch** [--list] | Afficher toutes les branches d’un dépôt |
| **git switch** <BRANCH> | Changer de branche |
| **git branch -**d / -D <BRANCH> | Supprimer une branche |
| **git branch** -m <OLD\_NAME> <NEW\_NAME> | Renommer une branche |
| **git branch** <BRANCH> [COMMIT\_HASH] | Créer une branche |
| **git switch** -c <BRANCH> [<COMMIT\_HASH>] | Créer une branche et y basculer |

**Merge**

|  |  |
| --- | --- |
| **git switch** <MAIN\_BRANCH> | Aller dans la branche principale |
| **git merge** <OTHER\_BRANCH> -m "MESSAGE" | Merge 2 branches |

**Rebase**

|  |  |
| --- | --- |
| **git switch** <OTHER\_BRANCH> | Aller dans la branche secondaire |
| **git rebase** <MAIN\_BRANCH> | Amène les modifications de main vers la branche actuelle |

**Reset**

|  |  |
| --- | --- |
| **git reset** –-soft <COMMIT\_HASH> | Revenir à ancien commit (garde le staging) |
| **git reset** –-hard <COMMIT\_HASH> | Revenir à ancien commit (supprime le staging) |
| **git restore** --staged <FILE | .> | Retirer un fichier du staging |
| **git restore** <FILE | .> | Revenir à un ancien commit |
| **git revert** <COMMIT\_HASH> | Remplace un commit par un nouveau |

**Remote**

|  |  |
| --- | --- |
| **git remote add** <REMOTE> <REPO\_URL> | Ajouter un remote |
| **git fetch** [<REMOTE>] | Récupérer les modifications d’un remote |
| **git merge** <REMOTE>/<BRANCH> | Synchroniser le contenu d’un remote localement |

**GitHub**

|  |  |
| --- | --- |
| **gh auth** login / logout | S’authentifier/se déco de GitHub avec GH CLI |
| **git remote add** origin <REPO\_URL> | Lier le dépôt local au remote sur GitHub |
| **git ls-remote** | Vérifier la liaison |
| **git push** [-u] origin <BRANCH> | Pousser les modifications vers le dépôt distant |
| **git push** origin <LOCAL\_BRANCH>:<REMOTE\_BRANCH> | Pousser une branche locale vers branche distante |
| **git push** origin :<REMOTE\_BRANCH> | Supprimer une branche distante (branche vide) |
| **git pull** [<REMOTE>/<BRANCH>] | Récupérer des modifis du remote dans dépôt local |

# Commandes avancées

**Reflog**

|  |  |
| --- | --- |
| **git reflog** | Affiche l’historique des déplacements récents |
| **git merge** <COMMITTISH> | Permet de merge avec le commitish au lieu du hash |

**Merge conflicts**

|  |  |
| --- | --- |
| **git merge** --abort | Arrêter un merge lors d’un conflit |
| **git checkout** –-ours <PATH> | Ecraser le fichier de la branche actuelle |
| **git checkout** –-theirs <PATH> | Ecraser le fichier de la branche en fusion |

**Rebase conflicts**

|  |  |
| --- | --- |
| **git rebase** --abort | Arrêter un rebase lors d’un conflit |
| **git rebase** --continue | Permet de terminer un rebase lors d’un conflit |
| **git checkout** –-ours <PATH> | Conserver les modifs de la branche en cours de rebase |
| **git checkout** –-theirs <PATH> | Remplacer les modifs de la branche cible (main) |
| **git reset** –-soft HEAD~1 | Annuler un commit (alors qu’il fallait –-continue) |

**RERERE**

|  |  |
| --- | --- |
| **git config** rerere.enabled {TRUE/FALSE} | Activer/désactiver RERERE |
| **Rm -rf .git/rr-cache** | Supprimer le dossier RERERE |

**Squash**

|  |  |
| --- | --- |
| **git rebase** -i HEAD~n | Démarrer le squashing |
| **git push** origin main --force | Forcer un push |

**Stash**

|  |  |
| --- | --- |
| **git stash** [-m “MESSAGE”] | Créer un stash |
| **git stash** list | Lister les stashes |
| **git stash** popstash@{n} | Revenir au stash et supprimer le stash du stack |
| **git stash** applystash@{n} | Revenir au stash SANS supprimer le stash du stack |
| **git stash** dreopstash@{n} | Supprimer un stash sans y retourner |

**Diff**

|  |  |
| --- | --- |
| **git diff** | Compare l’état actuel et le dernier commit |
| **git diff** HEAD~{n} | Comparer l’état actuel et un certain commit |
| **git diff** <COMMIT\_HASH\_1> <COMMIT\_HASH\_2> | Compare 2 commits |
| **git diff** --cached | Compare le staging avec la dernière validation |
| **git diff** HEAD | Compare l’arbre de travail avec la zone de staging |

**Cherry pick**

|  |  |
| --- | --- |
| **git cherry-pick** <COMMIT\_HASH> | Extraire un commit spécifique sans merge |
| **git cherry-pick -–**continue/--abort | Continuer/annuler à la fin d’un conflit |

**Bisect**

|  |  |
| --- | --- |
| **git bisect** start | Commencer le bisect |
| **git bisect** good <COMMIT\_HASH> | Marquer un commit fonctionnel |
| **git bisect** bad <COMMIT\_HASH> | Marquer un mauvais commit |
| **git bisect** good/bad | Déclarer le résultat du test |
| **git bisect** reset | Terminer le bisect |

**Worktree**

|  |  |
| --- | --- |
| **git worktree** list | Lister tous les worktrees d’un repo |
| **git worktree** add <PATH> [<BRANCH>] | Créer un linked worktree |
| **git worktree** remove <WORKTREE> | Supprimer un worktree |
| **git worktree p**rune | Nettoyer les références des worktrees supprimés |

**Tags**

|  |  |
| --- | --- |
| **git tag** | Lister les tags locaux |
| **git ls-remote** --tags | Lister les tags distants |
| **git tag** -a “TAG” <COMMIT\_HASH> -m “MESSAGE” | Créer un tag annoté |
| **git tag** -d “TAG\_NAME” | Supprimer un tag localement |
| **git push** origin–-delete <TAG\_NAME> | Supprimer un tag distant |
| **git push** origin–-tags | Pousser les tags vers le dépôt distant |

Une image contenant conception

Description générée automatiquement avec une confiance faible

Sources

Une image contenant symbole, clipart, créativité

Description générée automatiquement

# Références

Ben Straub, S. C. (2014). *Pro Git (2e édition).* Apress.

Git Documentation. (s.d.). *About Version Control*. Récupéré sur Git SCM: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Git Documentation. (s.d.). *Complete list of commands*. Récupéré sur Git SCM: https://git-scm.com/docs/git#\_git\_commands

Git Documentation. (s.d.). *Git Documentation*. Récupéré sur Git SCM: https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository

Git Documentation. (s.d.). *Git Tutorials and Training*. Récupéré sur Git SCM: https://git-scm.com/docs/gittutorial

GitHub. (s.d.). *About GitHub and Git*. Récupéré sur GitHub: https://docs.github.com/en/get-started/quickstart/about-github-and-git

GitHub. (s.d.). *Account and profile*. Récupéré sur GitHub Docs: https://docs.github.com/en/account-and-profile

GitHub. (s.d.). *Authentification*. Récupéré sur GitHub Docs: https://docs.github.com/en/authentication

GitHub. (s.d.). *Codespaces*. Récupéré sur GitHub Docs: https://docs.github.com/en/codespaces

GitHub. (s.d.). *Get started*. Récupéré sur GitHub Docs: https://docs.github.com/en/get-started

GitHub. (s.d.). *GitHub CLI*. Récupéré sur GitHub Docs: https://docs.github.com/en/github-cli

GitHub. (s.d.). *Pull requests*. Récupéré sur GitHub Docs: https://docs.github.com/en/pull-requests

GitHub. (s.d.). *Repositories*. Récupéré sur GitHub Docs: https://docs.github.com/en/repositories

Ogdolo LLC, dba Boot.dev. (s.d.). *Learn Git*. Récupéré sur Boot.Dev: www.boot.dev

Ogdolo LLC, dba Boot.dev. (s.d.). *Learn Git 2*. Récupéré sur Boot.Dev: www.boot.dev

W3School. (s.d.). *Git Tutorial*. Récupéré sur W3School: https://www.w3schools.com/git/