GUIDE PRATIQUE

Git au quotidien : le guide de référence pratique

Configuration, branches, rebase, résolution de conflits, récupération d'erreurs. Les commandes et workflows utilisés en équipe, expliqués clairement.

lapolaris.fr | 2026 | Édition gratuite

Tous niveaux

SOMMAIRE

Git, c'est le socle invisible de tout projet pro

Tu connais git add et git commit. Mais en entreprise, ça ne suffit pas. Ce guide te donne les vrais réflexes.

Chaque développeur professionnel utilise Git quotidiennement. Pourtant, la majorité des tutoriels s'arrêtent aux bases : init, add, commit, push. En situation réelle, tu dois gérer des branches, résoudre des conflits, réécrire un historique, retrouver un bug dans 2000 commits.

Ce guide couvre **les commandes et workflows que les développeurs expérimentés utilisent au quotidien**, celles qu'on apprend généralement sur le tas, en production, parfois dans la douleur.

Pas de théorie abstraite sur les DAG ou les objets internes de Git. Juste ce qui te sert quand tu codes, quand tu collabores, et quand quelque chose tourne mal.

Ce que tu vas apprendre

- Configurer Git avec les bons alias et les bonnes options pour gagner du temps chaque jour
- Maîtriser le workflow branche / commit / PR utilisé dans toutes les équipes pro
- Résoudre les conflits de merge calmement et efficacement
- Réécrire l'historique pour des commits propres et lisibles
- Utiliser stash, cherry-pick et bisect dans les situations qui les nécessitent
- Retrouver n'importe quel changement dans l'historique avec log, blame et reflog
- Corriger les erreurs courantes sans perdre de travail

PRÉ-REQUIS

Tu as déjà installé Git et tu sais faire un `git init`, `git add`, `git commit`. Si ce n'est pas le cas, commence par les bases avant de lire ce guide.

Configurer Git comme un pro

Avant d'écrire la moindre ligne de code, un dev expérimenté configure ses outils. Git ne fait pas exception.

Identité et options globales

La première chose à faire sur toute nouvelle machine. Ces commandes définissent qui tu es dans chaque commit :

TERMINAL

```
git config --global user.name "Ton Nom"
git config --global user.email "ton@email.com"
git config --global init.defaultBranch main
git config --global pull.rebase true
git config --global core.autocrlf input
```

> POURQUOI PULL.REBASE TRUE ?
>
> Par défaut, `git pull` crée un merge commit à chaque pull. Avec `pull.rebase true`, tes commits locaux sont replacés au-dessus des commits distants. L'historique reste linéaire et lisible.

Les alias qui changent la vie

Les développeurs expérimentés ne tapent pas les commandes complètes. Voici les alias les plus courants :

~/.GITCONFIG

```
[alias]
  st = status -sb
  co = checkout
  br = branch
  ci = commit
  lg = log --oneline --graph --decorate -20
  last = log -1 HEAD --stat
  unstage = reset HEAD --
  amend = commit --amend --no-edit
  wip = !git add -A && git commit -m "WIP"
```

Clé SSH : en finir avec les mots de passe

HTTPS demande ton mot de passe à chaque push. SSH le fait une seule fois. Génère ta clé et ajoute-la à GitHub :

TERMINAL

```
ssh-keygen -t ed25519 -C "ton@email.com" # Copie la clé publique

cat ~/.ssh/id_ed25519.pub # Colle-la dans GitHub > Settings > SSH keys
```

Outils visuels complémentaires

Le terminal est indispensable, mais certains outils graphiques accélèrent le travail sur les diffs et les merges :

GUI

GitKraken / Fork

Visualiser les branches, les merges et l'historique. Utile pour les projets complexes avec beaucoup de branches.

IDE

VS Code Git intégré

Gestion du staging, des diffs et des conflits directement dans l'éditeur. Suffisant pour 90% des cas.

DIFF

delta / diff-so-fancy

Remplace le diff par défaut de Git par un affichage coloré et lisible. S'installe en 2 minutes.

TUI

lazygit

Interface terminal interactive pour Git. Staging partiel, rebase, stash en quelques touches.

Le fichier .gitconfig complet d'un dev pro

~/.GITCONFIG

```
[core]
  editor = code --wait
  pager = delta
[merge]
  conflictstyle = diff3
[diff]
  colorMoved = default
[push]
  autoSetupRemote = true
```

PUSH.AUTOSETUPREMOTE

Avec cette option, `git push` crée automatiquement la branche distante si elle n'existe pas. Plus besoin de `git push -u origin ma-branche` la première fois. Disponible depuis Git 2.37.

Vérifier ta configuration

TERMINAL

```
git config --list --show-origin
# Affiche toutes les configs et leur source
# Utile pour débugger quand un comportement
# ne correspond pas à ce que tu attends
```

VERSION GIT REQUISE

Ce guide utilise `git switch` (Git 2.23+) et `push.autoSetupRemote` (Git 2.37+). Vérifie ta version avec `git --version` et mets à jour si nécessaire. Sur macOS : `brew upgrade git`. Sur Ubuntu : `sudo apt install git`.

Le cycle add, commit, push — mais en mieux

Tu connais les bases. Maintenant, apprends les réflexes qui font la différence entre un historique propre et un historique chaotique.

Toujours vérifier avant de commiter

Avant chaque commit, un dev pro fait systématiquement :

TERMINAL

```
git status          # Quels fichiers ont changé ?
git diff            # Qu'est-ce qui a changé exactement ?
git diff --staged   # Qu'est-ce qui va être commité ?
```

RÈGLE D'OR

Ne fais jamais `git add .` sans avoir d'abord fait `git status` et `git diff`. C'est comme ça qu'on commit des fichiers de debug, des clés API ou des `console.log` en production.

Le staging partiel : add -p

La commande la plus sous-estimée de Git. Elle te permet de choisir quels changements ajouter au commit, morceau par morceau :

TERMINAL

```
git add -p
# Git te montre chaque "hunk" (bloc de changements)
# y = ajouter, n = ignorer, s = découper plus fin
```

Résultat : un fichier peut contenir 3 modifications, et tu n'en commites que 2. Chaque commit reste atomique et cohérent.

Écrire de bons messages de commit

Un bon message de commit explique le **pourquoi**, pas le quoi. Le diff montre déjà le quoi.

MAUVAIS	BON
fix bug	fix: empêcher le double envoi du formulaire de paiement
update styles	ui: aligner le header sur la maquette mobile v2
wip	feat: ajouter le filtre par date sur la liste des commandes

CONVENTION CONVENTIONAL COMMITS

Préfixe tes commits : `feat:`, `fix:`, `refactor:`, `docs:`, `chore:`. Cette convention est utilisée par la majorité des projets open source et facilite la génération automatique de changelogs.

Synchroniser avec le distant

Le workflow de synchronisation en équipe suit toujours le même schéma :

TERMINAL

```
# Avant de commencer à travailler
git pull --rebase
# Après avoir commité
git push
# Si le push est rejeté (quelqu'un a push avant toi)
git pull --rebase && git push
```

Quand utiliser fetch vs pull

GIT FETCH

Récupérer sans appliquer

Télécharge les nouveaux commits du distant sans toucher à ton code. Tu peux ensuite inspecter avec `git log origin/main` avant de décider.

GIT PULL

Récupérer et appliquer

Fait un fetch + merge (ou rebase si configuré). Modifie directement tes fichiers. À utiliser quand tu es prêt à intégrer les changements.

Les commandes du quotidien résumées

COMMANDE	USAGE
`git status -sb`	Vue compacte des changements en cours
`git diff`	Voir les modifications non staged
`git diff --staged`	Voir ce qui sera commité
`git add -p`	Staging interactif, hunk par hunk
`git commit -m "msg"`	Commiter avec un message court
`git commit`	Ouvre l'éditeur pour un message détaillé
`git pull --rebase`	Synchroniser proprement
`git push`	Envoyer ses commits sur le distant

RÉFLEXE À ADOPTER

Commite souvent, pushe régulièrement. Un commit toutes les 30 minutes de travail est un bon rythme. Des petits commits atomiques sont plus faciles à reviewer, à reverter et à comprendre.

Branches : créer, naviguer, merger

Les branches sont le coeur du travail collaboratif avec Git. Chaque feature, chaque fix a sa propre branche.

Créer et changer de branche

TERMINAL

```
# Créer une branche et y aller directement
git checkout -b feat/ajout-panier
# Ou avec la syntaxe moderne (Git 2.23+)
git switch -c feat/ajout-panier

# Changer de branche
git switch main
# Lister les branches
git branch         # locales
git branch -a      # locales + distantes
```

Conventions de nommage

En équipe, les branches suivent des conventions. Les plus courantes :

- `feat/nom-feature` — nouvelle fonctionnalité
- `fix/description-bug` — correction de bug
- `refactor/ce-qui-change` — refactoring sans changement fonctionnel
- `hotfix/urgence` — correction critique en production
- `chore/maintenance` — tâches techniques (deps, CI, etc.)

Merge vs Rebase : le grand débat

GIT MERGE

Fusionner avec historique

Crée un commit de merge. Préserve l'historique exact des branches. Lisible mais peut devenir bruyant.

GIT REBASE

Rejouer par-dessus

Replace tes commits au-dessus de la branche cible. Historique linéaire et propre. Ne jamais rebase une branche partagée.

RÈGLE SIMPLE

Rebase pour mettre à jour ta branche locale avec main. Merge (via PR) pour intégrer ta branche dans main. C'est le workflow utilisé par la plupart des équipes professionnelles.

Mettre à jour sa branche avec main

TERMINAL

```
# Sur ta branche feature
git fetch origin
git rebase origin/main
# Résoudre les conflits s'il y en a, puis :
git rebase --continue
```

Supprimer les branches inutiles

Après chaque merge, supprime ta branche. L'accumulation de branches mortes rend le dépôt illisible :

TERMINAL

```
# Supprimer une branche locale mergée
git branch -d feat/ajout-panier
# Supprimer la branche distante
git push origin --delete feat/ajout-panier
# Nettoyer les refs distantes supprimées
git fetch --prune
```

ASTUCE : NETTOYAGE EN MASSE

Pour supprimer toutes les branches locales déjà mergées dans main : `git branch --merged main | grep -v main | xargs git branch -d`. À exécuter régulièrement.

Le workflow Feature Branch en résumé

- [ ] Créer une branche depuis main avec un nom descriptif
- [ ] Travailler et commiter sur cette branche
- [ ] Rebaser régulièrement sur main pour rester à jour
- [ ] Pusher et ouvrir une Pull Request
- [ ] Après review et merge, supprimer la branche

1	< 3j	< 400
branche = 1 sujet	durée de vie idéale d'une branche	lignes modifiées par PR pour une bonne review

Résoudre les conflits sans paniquer

Les conflits ne sont pas des erreurs. C'est Git qui te dit : "deux personnes ont modifié le même endroit, choisis."

Anatomie d'un conflit

Quand Git ne peut pas fusionner automatiquement, il insère des marqueurs dans le fichier :

FICHIER EN CONFLIT

```
<<<<<<< HEAD
const prix = calculerPrix(produit);
=======
const prix = getPrixTTC(produit, tva);
>>>>>>> feat/calcul-tva
```

- `<<<<<<< HEAD` — ta version actuelle (la branche sur laquelle tu es)
- `=======` — séparateur entre les deux versions
- `>>>>>>> feat/...` — la version de la branche entrante

Les étapes pour résoudre

TERMINAL

```
# 1. Identifier les fichiers en conflit
git status
# 2. Ouvrir chaque fichier, choisir la bonne version
# 3. Supprimer les marqueurs <<< === >>>
# 4. Ajouter les fichiers résolus
git add fichier-resolu.js
# 5. Continuer le merge ou rebase
git merge --continue   # ou git rebase --continue
```

CONFIGURER VS CODE COMME OUTIL DE MERGE

VS Code gère les conflits nativement avec des boutons "Accept Current", "Accept Incoming", "Accept Both". Pour en faire ton outil par défaut : `git config --global merge.tool vscode` et `git config --global mergetool.vscode.cmd 'code --wait $MERGED'`. Lance ensuite `git mergetool` lors d'un conflit.

EN CAS DE PANIQUE

Tu peux toujours annuler un merge ou un rebase en cours avec `git merge --abort` ou `git rebase --abort`. Tu retrouves l'état exact d'avant. Aucun risque de perdre du travail.

Réécrire l'historique proprement

Un historique propre n'est pas un luxe. C'est ce qui permet à ton équipe de comprendre le projet dans 6 mois.

Modifier le dernier commit

TERMINAL

```
# Modifier le message du dernier commit
git commit --amend -m "nouveau message"

# Ajouter des fichiers oubliés au dernier commit
git add fichier-oublie.js
git commit --amend --no-edit
```

ATTENTION

`--amend` réécrit le dernier commit. Si tu l'as déjà pushé, il faudra faire un `git push --force-with-lease`. Ne fais ça que sur ta propre branche, jamais sur main.

Le rebase interactif

C'est l'outil de réécriture le plus puissant de Git. Il te permet de modifier les N derniers commits :

TERMINAL

```
git rebase -i HEAD~5
# Ouvre l'éditeur avec les 5 derniers commits :
pick a1b2c3d feat: ajouter le panier
pick e4f5g6h fix: corriger le total
pick i7j8k9l wip
pick m0n1o2p fix: typo
pick q3r4s5t feat: ajouter le paiement
```

Les actions du rebase interactif

ACTION	EFFET
`pick`	Garder le commit tel quel
`reword`	Garder le commit mais modifier le message
`squash`	Fusionner avec le commit précédent (combiner les messages)
`fixup`	Fusionner avec le précédent (garder seulement son message)
`drop`	Supprimer le commit

Exemple concret : nettoyer avant une PR

Tu as 5 commits dont 2 "wip" et 1 "fix typo". Avant d'ouvrir ta PR, tu nettoies :

AVANT

```
feat: ajouter le panier
wip
fix: corriger le total
fix: typo dans le calcul
feat: ajouter le paiement
```

APRÈS REBASE -I (SQUASH + FIXUP)

```
feat: ajouter le panier avec calcul du total
feat: ajouter le paiement
```

L'historique est maintenant propre : chaque commit raconte une histoire claire et complète.

Le commit fixup automatique

Git propose un raccourci pour préparer un commit qui sera fusionné automatiquement :

TERMINAL

```
# Créer un commit fixup lié à un commit existant
git commit --fixup=a1b2c3d
# Plus tard, appliquer tous les fixups automatiquement
git rebase -i --autosquash HEAD~5
```

ACTIVER AUTOSQUASH PAR DÉFAUT

Ajoute `git config --global rebase.autoSquash true` pour que les commits fixup soient automatiquement placés au bon endroit lors d'un rebase interactif.

Force push en sécurité

Après un rebase, tu dois force push. Mais `--force` est dangereux car il écrase tout. Utilise toujours :

TERMINAL

```
git push --force-with-lease
# Refuse le push si quelqu'un d'autre a pushé
# entre-temps. Filet de sécurité indispensable.
```

RÈGLE ABSOLUE

Ne réécris jamais l'historique d'une branche partagée (main, develop, staging). Si tu force push sur une branche partagée, les commits de tes collègues deviennent orphelins : Git leur refusera les prochains push, et ils devront reconstruire leur historique manuellement. Le rebase interactif est réservé à tes branches personnelles, avant le merge.

Stash, cherry-pick et bisect

Stash et cherry-pick sont des réflexes du quotidien. Bisect est rare, mais irremplaçable pour chasser un bug dans un long historique.

git stash : mettre de côté temporairement

Tu travailles sur une feature et on te demande un hotfix urgent. Stash sauvegarde tes modifications en cours :

TERMINAL

```
# Sauvegarder les modifications en cours
git stash
# Avec un message descriptif (recommandé)
git stash push -m "wip: formulaire inscription"
# Lister les stashs
git stash list
# Récupérer le dernier stash
git stash pop
# Récupérer un stash spécifique
git stash apply stash@{2}
```

> STASH INCLUANT LES FICHIERS NON TRACKÉS
>
> Par défaut, `git stash` ignore les fichiers non suivis. Ajoute `-u` pour les inclure : `git stash -u`.

git cherry-pick : prendre un commit d'ailleurs

Tu veux appliquer un commit spécifique d'une autre branche sans merger toute la branche :

TERMINAL

```
# Appliquer un commit précis sur ta branche
git cherry-pick a1b2c3d
# Appliquer sans commiter (pour modifier avant)
git cherry-pick a1b2c3d --no-commit
```

Cas d'usage typique : un fix est sur une branche feature, mais tu en as besoin sur main tout de suite.

git bisect : trouver le commit qui a cassé

Le bug n'était pas là lundi. Il est là aujourd'hui. Bisect fait une recherche binaire dans l'historique :

TERMINAL

```
git bisect start
git bisect bad                  # le commit actuel est cassé
git bisect good abc123          # ce commit était OK
git bisect good                 # ou : git bisect bad — Git checkout le commit du milieu
git bisect reset                # revenir à la normale une fois le coupable trouvé
```

git bisect : aller plus loin

Bisect peut s'automatiser. Si tu as un script qui retourne 0 (OK) ou 1 (cassé), Git fait la recherche binaire tout seul :

TERMINAL

```
# Bisect automatisé avec un script de test
git bisect start
git bisect bad HEAD
git bisect good v1.4.2
git bisect run npm test
# Git checkout chaque commit, lance npm test,
# et identifie automatiquement le commit coupable.
git bisect reset   # Revenir à la normale
```

COMBIEN DE COMMITS BISECT VÉRIFIE-T-IL ?

Bisect utilise une recherche binaire : sur 1000 commits suspects, il n'en testera que ~10. C'est son atout majeur face à une inspection manuelle.

Limites et bonnes pratiques du stash

À SAVOIR

Stash n'est pas un commit

Les entrées de stash peuvent être perdues si tu supprimes la branche associée ou nettoies le dépôt (`git gc`). Ne stocke pas du travail important longtemps dans le stash.

ALTERNATIVE

WIP commit

Pour sauvegarder un travail en cours de façon plus robuste, crée un commit temporaire : `git add -A && git commit -m "WIP"` . Puis `git reset --soft HEAD~1` pour le défaire.

cherry-pick : cas d'usage typiques

SITUATION	SOLUTION
Un fix est sur feat/x mais urgent sur main	`git cherry-pick <hash>`
Backporter un fix vers une release ancienne	`git cherry-pick <hash>`
Appliquer sans commiter (pour ajuster)	`git cherry-pick --no-commit <hash>`

CHERRY-PICK ET DUPLICATS

Cherry-pick crée un nouveau commit avec un hash différent. Si la branche source est ensuite mergée, Git peut introduire des doublons. Préfère merger la branche si tu en as besoin en entier.

Git log et la recherche dans l'historique

Savoir naviguer dans l'historique, c'est ce qui te permet de comprendre pourquoi le code est comme il est.

git log : les options qui comptent

TERMINAL

```
# Log compact avec graphe
git log --oneline --graph --decorate -20
# Log d'un fichier spécifique
git log --oneline -- src/utils/prix.js
# Log avec les diffs
git log -p -3    # Les 3 derniers commits avec le code
# Chercher dans les messages de commit
git log --grep="panier" --oneline
# Commits d'un auteur
git log --author="Marie" --oneline --since="2 weeks ago"
```

git blame : qui a écrit cette ligne ?

TERMINAL

```
git blame src/utils/prix.js # Voir qui a modifié chaque ligne
git blame -L 10,25 src/utils/prix.js # Limiter à un intervalle de lignes
```

> BLAME N'EST PAS POUR BLÂMER
>
> Le but n'est pas de trouver un coupable. C'est de comprendre le contexte : qui a changé cette ligne, dans quel commit, pour quelle raison. Puis `git show <hash>` pour voir le commit complet.

git show : inspecter un commit

TERMINAL

```
git show a1b2c3d # Voir le contenu complet d'un commit
git show a1b2c3d:src/utils/prix.js # Voir un fichier à un commit donné
```

git reflog : le filet de sécurité ultime

Le reflog enregistre tout ce que tu fais avec HEAD. Même après un reset --hard, tu peux retrouver tes commits :

TERMINAL

```
git reflog
# Affiche l'historique de tous les mouvements de HEAD. Tu peux revenir à n'importe quel état avec :
git reset --hard HEAD@{3}
```

Pull requests et code review

La PR n'est pas juste un bouton "Merge". C'est le moment où ton code est lu, discuté et amélioré par ton équipe.

Anatomie d'une bonne PR

- **Titre clair** : ce que la PR fait, en une ligne. Ex : "feat: ajouter le filtre par date"
- **Description** : le pourquoi, le contexte, les choix techniques. Pas un roman, mais assez pour comprendre sans lire le code
- **Taille raisonnable** : moins de 400 lignes modifiées. Au-delà, la qualité de la review chute drastiquement
- **Un sujet = une PR** : ne mélange pas un fix et un refactor dans la même PR
- **Tests** : si le projet en a, ta PR doit les passer. Bonus si tu en ajoutes

Le workflow PR avec GitHub CLI

TERMINAL

```
# Créer une PR depuis le terminal
gh pr create --title "feat: filtre par date" \
  --body "Ajoute un filtre par date sur la liste."

# Voir les PRs ouvertes
gh pr list
# Checkout une PR pour la tester localement
gh pr checkout 42
# Merger une PR
gh pr merge 42 --squash
```

Faire une bonne code review

REVIEWER

Ce qu'on cherche

Bugs potentiels, lisibilité, nommage, cas limites, sécurité. Pas le style personnel.

AUTEUR

Comment répondre

Remercier les retours, expliquer ses choix sans se justifier. Si le commentaire est pertinent, corriger.

CONVENTION D'ÉQUIPE

Beaucoup d'équipes utilisent des préfixes dans les commentaires de review : `nit:` (détail mineur), `question:` (je ne comprends pas), `suggestion:` (idée d'amélioration), `blocker:` (à corriger avant merge).

Les erreurs courantes et comment les corriger

Tout le monde fait des erreurs avec Git. Ce qui compte, c'est de savoir les corriger sans perdre de travail.

J'ai commité sur la mauvaise branche

TERMINAL

```
# Déplacer le dernier commit vers une autre branche
git branch nouvelle-branche    # Sauver le commit
git reset --hard HEAD~1        # Enlever de la branche actuelle
git switch nouvelle-branche    # Aller sur la bonne branche
```

J'ai pushé quelque chose que je n'aurais pas dû

TERMINAL

```
# Créer un commit qui annule les changements
git revert a1b2c3d
# Revert crée un nouveau commit d'annulation. L'historique est préservé, rien n'est effacé
```

> REVERT VS RESET
>
> `revert` crée un nouveau commit qui annule. Sûr pour les branches partagées. `reset` efface des commits de l'historique. À réserver aux branches personnelles non pushées.

J'ai fait un reset --hard et j'ai perdu du travail

TERMINAL

```
git reflog # Le reflog garde tout pendant 90 jours
# Trouver le hash du commit perdu, puis :
git reset --hard HEAD@{n}
# Ou créer une branche pour le récupérer :
git branch recovery HEAD@{n}
```

Les 3 niveaux de reset

COMMANDE	STAGING	FICHIERS
`git reset --soft HEAD~1`	Conservé	Conservés
`git reset HEAD~1`	Vidé	Conservés
`git reset --hard HEAD~1`	Vidé	Effacés

J'ai ajouté un fichier sensible (.env, clé API)

TERMINAL

```
# Si pas encore pushé : retirer du staging
git reset HEAD .env
# Ajouter au .gitignore immédiatement
echo ".env" >> .gitignore
# Si déjà pushé : CHANGER LES CLÉS IMMÉDIATEMENT
# Puis installer git-filter-repo (outil officiel Git) :
# pip install git-filter-repo
git filter-repo --path .env --invert-paths --force
# Puis forcer le push sur toutes les branches :
git push origin --force --all
```

URGENCE SÉCURITÉ

Si tu as pushé des clés API ou des mots de passe, les supprimer de l'historique ne suffit pas. Considère-les compromis. Révoque-les et génère de nouvelles clés immédiatement. `git filter-repo` remplace l'ancien `git filter-branch`, déprécié et dangereux.

J'ai fait un merge que je veux annuler

TERMINAL

```
# Annuler un merge commit (garder l'historique)
git revert -m 1 <hash-du-merge-commit>
# -m 1 indique de garder le parent 1 (ta branche)
```

Résumé : quelle commande pour quelle erreur

SITUATION	COMMANDE
Annuler le staging d'un fichier	`git reset HEAD fichier`
Annuler les modifications d'un fichier	`git checkout -- fichier`
Annuler le dernier commit (garder les fichiers)	`git reset --soft HEAD~1`
Annuler un commit pushé	`git revert <hash>`
Récupérer un commit perdu	`git reflog + git reset`
Annuler un merge en cours	`git merge --abort`
Annuler un rebase en cours	`git rebase --abort`

MANTRA DU DEV PRO

Avec Git, presque rien n'est irréversible. Si tu as commité, le reflog le garde 90 jours. Respire, cherche, récupère.

.gitignore et hygiène de dépôt

Un dépôt propre, c'est un dépôt où on ne trouve que du code source. Pas de node_modules, pas de .DS_Store, pas de builds.

Le .gitignore essentiel

.GITIGNORE

```
# Dépendances
node_modules/
vendor/
__pycache__/

# Environnement
.env
.env.local
*.log

# IDE
.vscode/settings.json
.idea/
*.swp

# Build
dist/
build/
*.min.js

# OS
.DS_Store
Thumbs.db
```

GITIGNORE GLOBAL

Les fichiers liés à ton OS ou ton IDE (`.DS_Store`, `.idea/`) doivent être dans ton gitignore global, pas dans chaque projet : `git config --global core.excludesfile ~/.gitignore_global`

Retirer un fichier déjà suivi

TERMINAL

```
# Le fichier est dans .gitignore mais Git le suit encore
git rm --cached fichier.log
# Pour un dossier entier
git rm -r --cached node_modules/
git commit -m "chore: retirer les fichiers ignorés du tracking"
```

Bonnes pratiques d'hygiène

- Génère ton `.gitignore` de base sur **gitignore.io** en indiquant ton stack (Node, Python, macOS...)
- Ajoute `.env.example` au dépôt : un fichier d'exemple avec les clés vides, pour documenter les variables nécessaires sans exposer les valeurs
- Utilise `git check-ignore -v fichier` pour débugger pourquoi un fichier est ou n'est pas ignoré
- Nettoie les fichiers non trackés avec `git clean -fd` (attention : irréversible sans `-n` pour prévisualiser)

VÉRIFIER AVANT DE PUBLIER

Avant d'ouvrir une PR ou de publier un dépôt, fais `git log --oneline --all` et `git status` pour t'assurer qu'aucun fichier sensible ne traîne dans l'historique ou le staging.

CONCLUSION

Git se maîtrise par la pratique quotidienne

Tu n'as pas besoin de tout retenir. Tu as besoin de pratiquer, de faire des erreurs, et de savoir où chercher.

Les 5 réflexes du dev pro avec Git

Toujours vérifier avant de commiter. Toujours travailler sur une branche. Toujours écrire des messages clairs. Toujours nettoyer avant de merger. Toujours savoir annuler.

Les signaux que tu maîtrises Git

Tu résous un conflit sans stress

Les marqueurs ne te font plus peur, tu comprends les deux versions.

Tu nettoies ton historique

Rebase interactif, squash et fixup font partie de ton workflow.

Tu retrouves tout

Log, blame et reflog n'ont plus de secret pour toi.

Ressources pour aller plus loin

LIVRE

Pro Git (gratuit)

Le livre de référence, écrit par les créateurs de GitHub. Disponible gratuitement en ligne et en français.

INTERACTIF

Learn Git Branching

Visualiser les branches et le rebase de manière interactive. Le meilleur outil pour comprendre Git.

RÉFÉRENCE

Oh Shit, Git!?!

Solutions rapides aux erreurs Git les plus courantes. En anglais, direct et efficace.

CHEAT SHEET

GitHub Git Cheat Sheet

Fiche de référence officielle GitHub avec les commandes les plus utilisées.

Formations Git chez LaPolaris

Maîtrise Git avec des exercices pratiques, des mises en situation d'équipe et du code review.

Voir les formations

CONFIGURATION

`git config --global user.name "Nom"`	Définir son identité
`git config --global pull.rebase true`	Pull en mode rebase

WORKFLOW QUOTIDIEN

`git status -sb`	État compact
`git diff / git diff --staged`	Voir les changements
`git add -p`	Staging interactif
`git commit -m "msg"`	Commiter
`git pull --rebase`	Synchroniser
`git push`	Envoyer

BRANCHES

`git switch -c nom`	Créer et aller sur une branche
`git branch -d nom`	Supprimer une branche mergée
`git rebase origin/main`	Mettre à jour sa branche

HISTORIQUE

`git log --oneline --graph -20`	Historique compact
`git blame fichier`	Qui a écrit quoi
`git reflog`	Historique de HEAD
`git show <hash>`	Détail d'un commit

RÉÉCRITURE ET CORRECTION

`git commit --amend`	Modifier le dernier commit
`git rebase -i HEAD~N`	Réécrire les N derniers commits
`git stash / git stash pop`	Mettre de côté / récupérer
`git cherry-pick <hash>`	Appliquer un commit d'ailleurs
`git revert <hash>`	Annuler un commit (safe)
`git reset --soft HEAD~1`	Défaire le dernier commit (garder les fichiers)
`git push --force-with-lease`	Force push sécurisé