Guide Expert Git & GitHub - Travail Collaboratif

Table des matières

Concepts Fondamentaux

Les 4 Zones de Git

```
Working Directory (Zone de travail)
        ↓ git add
Staging Area (Index)
        ↓ git commit
Local Repository (Dépôt local)
        ↓ git push
Remote Repository (Dépôt distant - GitHub)
```

Vocabulaire Essentiel

Terme	Définition	Impact
Working Directory	Répertoire où vous modifiez les fichiers	Modifications non trackées
Staging Area (Index)	Zone tampon avant commit	Fichiers prêts à être committés
HEAD	Pointeur vers le dernier commit de la branche actuelle	Référence de position
Origin	Nom par défaut du dépôt distant	Lien avec GitHub
Master/Main	Branche principale de production	Code stable déployable
Feature Branch	Branche de développement d'une fonctionnalité	Isolation du travail
Merge	Fusion de deux branches	Intégration des modifications

Terme	Définition	Impact
Rebase	Réécriture de l'historique	Historique linéaire
Pull Request (PR)	Demande de fusion sur GitHub	Review de code avant merge
Fork	Copie d'un repo vers votre compte	Contribution externe
Stash	Sauvegarde temporaire des modifications	Nettoyage rapide du working directory

---

Configuration Initiale

Configuration Utilisateur

bash

```
# Configuration globale (pour tous les projets)
git config --global user.name "Votre Nom"
git config --global user.email "votre.email@example.com"

# Configuration locale (pour un projet spécifique)
git config --local user.name "Nom Professionnel"
git config --local user.email "pro@company.com"

# Vérifier la configuration
git config --list
git config user.name
git config user.email
```

Impact : Identifie l'auteur de chaque commit. Essentiel pour la traçabilité.

Configuration Éditeur et Outils

bash

```
# Définir l'éditeur par défaut
git config --global core.editor "code --wait"  # VSCode
git config --global core.editor "vim"          # Vim
git config --global core.editor "nano"         # Nano

# Configuration des couleurs
git config --global color.ui auto

# Configuration des fins de ligne (important Windows/Linux)
git config --global core.autocrlf true   # Windows
git config --global core.autocrlf input  # Mac/Linux

# Définir le nom de la branche principale
git config --global init.defaultBranch main
```

Authentification GitHub

bash

```
# Configurer l'authentification SSH (RECOMMANDÉ)
ssh-keygen -t ed25519 -C "votre.email@example.com"
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519

# Tester la connexion SSH
ssh -T git@github.com

# Ou utiliser HTTPS avec Personal Access Token (PAT)
# GitHub Settings → Developer settings → Personal access tokens → Generate new token
```

Impact : Permet de push/pull sans entrer le mot de passe à chaque fois.

Workflow Quotidien

1. Cloner un Dépôt

bash

```
# Cloner avec HTTPS
git clone https://github.com/username/repo.git

# Cloner avec SSH (recommandé)
git clone git@github.com:username/repo.git

# Cloner une branche spécifique
git clone -b feature-branch git@github.com:username/repo.git

# Cloner avec un nom de dossier personnalisé
git clone git@github.com:username/repo.git mon-dossier
```

Impact : Crée une copie locale complète du projet avec tout l'historique.

2. Vérifier l'État

bash

```
# Voir l'état des fichiers
git status

# Version courte
git status -s

# Voir les différences non stagées
git diff

# Voir les différences stagées
git diff --staged
# ou
git diff --cached
```

Impact : Comprendre exactement ce qui a changé avant de commiter.

3. Ajouter des Modifications

bash

```bash
# Ajouter un fichier spécifique
git add fichier.txt

# Ajouter plusieurs fichiers
git add fichier1.txt fichier2.txt

# Ajouter tous les fichiers modifiés et nouveaux
git add .

# Ajouter tous les fichiers (y compris supprimés)
git add -A

# Ajouter de manière interactive
git add -p
# Vous permet de choisir quelles parties d'un fichier ajouter

# Ajouter tous les fichiers d'un type
git add *.js
git add src/
```

Impact : Déplace les fichiers du Working Directory vers le Staging Area. Aucun impact sur le dépôt tant que vous n'avez pas commit.

4. Commiter les Modifications

```bash
# Commit avec message
git commit -m "feat: ajout de la fonctionnalité X"

# Commit avec message multi-lignes
git commit -m "feat: ajout de la fonctionnalité X" -m "Description détaillée de la fonctionnalité"

# Commit en ajoutant automatiquement tous les fichiers modifiés (PAS les nouveaux)
git commit -am "fix: correction du bug Y"

# Modifier le dernier commit (message ou contenu)
git commit --amend -m "Nouveau message"

# Amend sans changer le message
git commit --amend --no-edit
```

Impact : Enregistre un snapshot permanent dans l'historique local. Pas encore visible sur GitHub.

Convention de Nommage des Commits (Conventional Commits)

```bash
```

feat: Nouvelle fonctionnalité
fix: Correction de bug
docs: Documentation
style: Formatage, point-virgules manquants, etc.
refactor: Refactoring du code
test: Ajout ou modification de tests
chore: Maintenance, mise à jour dépendances
perf: Amélioration des performances
ci: Modification CI/CD
build: Modification système de build

Exemples :

```bash
git commit -m "feat: ajout de l'authentification JWT"
git commit -m "fix: correction de la connexion RDS"
git commit -m "docs: mise à jour du README avec instructions Docker"
git commit -m "refactor: optimisation des requêtes SQL"
```

5. Pousser vers GitHub

```bash
# Pousser la branche actuelle vers origin
git push origin feature-branch

# Pousser et définir la branche upstream (première fois)
git push -u origin feature-branch
# Ensuite, vous pouvez juste faire : git push

# Pousser toutes les branches
git push --all origin

# Pousser les tags
git push --tags

# Forcer le push (⚠️ DANGEREUX)
git push --force origin feature-branch

# Force push plus sûr
git push --force-with-lease origin feature-branch
```

Impact :

- `git push` : Envoie vos commits locaux vers GitHub. Visible par toute l'équipe.

- `--force` : **ÉCRASE** l'historique distant. Peut faire perdre le travail des autres ! À utiliser UNIQUEMENT si vous êtes seul sur la branche.
- `--force-with-lease` : Plus sûr, refuse si quelqu'un a pushé entre temps.

6. Récupérer les Modifications

```bash
# Récupérer ET fusionner les changements distants
git pull origin feature-branch

# Récupérer les informations sans fusionner
git fetch origin

# Fetch toutes les branches
git fetch --all

# Pull avec rebase (recommandé pour historique propre)
git pull --rebase origin feature-branch

# Voir les différences après fetch
git diff HEAD..origin/feature-branch
```

Impact :

- `fetch` : Télécharge les commits distants mais ne modifie RIEN localement. Sans danger.
- `pull` = `fetch` + `merge` : Fusionne automatiquement. Peut créer un commit de merge.
- `pull --rebase` : Réapplique vos commits par-dessus les commits distants. Historique linéaire.

Gestion des Branches

Commandes de Base

```bash
```

```
# Lister toutes les branches locales
git branch

# Lister toutes les branches (locales + distantes)
git branch -a

# Lister les branches distantes uniquement
git branch -r

# Créer une nouvelle branche
git branch nouvelle-branche

# Créer et basculer vers une nouvelle branche
git checkout -b feature/nouvelle-fonctionnalite
# ou avec la nouvelle syntaxe
git switch -c feature/nouvelle-fonctionnalite

# Basculer vers une branche existante
git checkout feature-branch
# ou
git switch feature-branch

# Créer une branche à partir d'un commit spécifique
git branch nouvelle-branche abc123

# Renommer la branche actuelle
git branch -m nouveau-nom

# Renommer une autre branche
git branch -m ancien-nom nouveau-nom

# Supprimer une branche locale (si déjà mergée)
git branch -d feature-branch

# Forcer la suppression d'une branche locale
git branch -D feature-branch

# Supprimer une branche distante
git push origin --delete feature-branch
```

Impact :

- Créer une branche : **Aucun impact** sur le reste du projet. Travail isolé.
- Supprimer une branche mergée : Sans danger, le travail est préservé dans main.
- Supprimer avec `-D` : **PERTE DÉFINITIVE** si le travail n'est pas mergé ailleurs.

Workflow GitFlow

```
main (production)
  ↓
develop (développement)
  ↓
feature/* (fonctionnalités)
hotfix/* (corrections urgentes)
release/* (préparation versions)
```

bash

```bash
# Créer une branche feature depuis develop
git checkout develop
git pull origin develop
git checkout -b feature/auth-system

# Travailler, commiter...
git add .
git commit -m "feat: ajout système d'authentification"

# Mettre à jour depuis develop régulièrement
git checkout develop
git pull origin develop
git checkout feature/auth-system
git merge develop

# Ou avec rebase (préféré)
git rebase develop

# Une fois terminé, merger dans develop
git checkout develop
git merge feature/auth-system
git push origin develop

# Supprimer la branche feature
git branch -d feature/auth-system
git push origin --delete feature/auth-system
```

Workflow GitHub Flow (Plus Simple)

```
main
  ↓
feature-branches
```

↓
Pull Request → Review → Merge → main

```bash
# 1. Créer une branche depuis main
git checkout main
git pull origin main
git checkout -b fix/database-connection

# 2. Travailler et pousser régulièrement
git add .
git commit -m "fix: correction timeout connexion DB"
git push origin fix/database-connection

# 3. Créer une Pull Request sur GitHub

# 4. Après approbation et merge, nettoyer
git checkout main
git pull origin main
git branch -d fix/database-connection
```

Collaboration en Équipe

Synchronisation Continue

```bash
# Workflow recommandé AVANT de commencer à travailler
git checkout main
git pull origin main
git checkout -b feature/my-work

# Workflow recommandé PENDANT le travail (plusieurs fois par jour)
git fetch origin
git rebase origin/main  # Ou merge selon votre préférence

# Workflow recommandé AVANT de pusher
git fetch origin
git rebase origin/feature/my-work
git push origin feature/my-work
```

Mise à Jour d'une Branche Feature depuis Main

```bash
```

```bash
# Option 1 : Merge (crée un commit de merge)
git checkout feature/my-branch
git merge main

# Option 2 : Rebase (historique linéaire - RECOMMANDÉ)
git checkout feature/my-branch
git rebase main

# Si des conflits apparaissent pendant le rebase
# 1. Résoudre les conflits dans les fichiers
# 2. Ajouter les fichiers résolus
git add .
# 3. Continuer le rebase
git rebase --continue

# Annuler le rebase si ça tourne mal
git rebase --abort
```

Impact :

- **Merge** : Conserve l'historique complet. Crée un commit de merge. Historique en "diamant".
- **Rebase** : Réécrit l'historique comme si vous aviez travaillé sur la dernière version de main. Historique linéaire, plus propre.
- ⚠️ **NE JAMAIS REBASER** une branche publique partagée avec d'autres ! Seulement vos branches personnelles.

Travailler sur le Code d'un Collègue

bash

```bash
# Récupérer une branche d'un collègue
git fetch origin
git checkout -b feature-colleague origin/feature-colleague

# Ou plus court
git checkout feature-colleague  # Git créera automatiquement la branche locale

# Voir toutes les branches distantes disponibles
git branch -r

# Mettre à jour la branche de votre collègue
git pull origin feature-colleague

# Ajouter vos modifications et pousser
git add .
git commit -m "feat: amélioration de la feature"
git push origin feature-colleague
```

Code Review avec Pull Requests

```bash
# 1. Créer votre branche et pousser
git checkout -b feature/new-api
git add .
git commit -m "feat: ajout API REST"
git push -u origin feature/new-api

# 2. Sur GitHub : Create Pull Request

# 3. Si des changements sont demandés
git add .
git commit -m "refactor: prise en compte des remarques"
git push origin feature/new-api  # La PR se met à jour automatiquement

# 4. Squash des commits avant merge (optionnel)
git rebase -i HEAD~3  # Pour les 3 derniers commits
# Remplacer "pick" par "squash" pour fusionner les commits
git push --force-with-lease origin feature/new-api

# 5. Après merge de la PR sur GitHub
git checkout main
git pull origin main
git branch -d feature/new-api
```

Résolution de Conflits

Identifier un Conflit

```bash
# Après un merge ou rebase, Git vous indique :
git status

# Fichiers en conflit marqués comme "both modified"
```

Structure d'un Conflit

```
<<<<<<< HEAD (votre version actuelle)
const dbHost = "localhost";
=======
const dbHost = "aurora.cluster.amazonaws.com";
>>>>>>> feature-branch (version entrante)
```

Résoudre un Conflit

```bash
# 1. Ouvrir les fichiers en conflit et choisir la bonne version
# Supprimer les marqueurs <<<<<<<, =======, >>>>>>>

# 2. Tester que le code fonctionne

# 3. Ajouter les fichiers résolus
git add fichier-resolu.js

# 4. Si vous étiez en train de merger
git commit -m "Merge: résolution conflits"

# 5. Si vous étiez en train de rebaser
git rebase --continue

# Annuler la résolution et recommencer
git merge --abort  # Pour un merge
git rebase --abort  # Pour un rebase
```

Outils de Résolution de Conflits

```bash
```

```bash
# Lancer un outil de merge visuel
git mergetool

# Configurer VSCode comme outil de merge
git config --global merge.tool vscode
git config --global mergetool.vscode.cmd 'code --wait $MERGED'

# Voir les différentes versions du fichier
git show :1:fichier.js  # Version de base commune
git show :2:fichier.js  # Version locale (HEAD)
git show :3:fichier.js  # Version distante (incoming)
```

Stratégies pour Éviter les Conflits

1. **Communiquer** : Informez l'équipe des fichiers que vous modifiez
2. **Pull régulièrement** : Récupérez les changements plusieurs fois par jour
3. **Commits atomiques** : Petits commits fréquents plutôt que gros commits rares
4. **Modularité** : Travaillez sur des fichiers/fonctions différents
5. **Branches courtes** : Fusionnez rapidement (feature branches de 1-3 jours max)

Commandes Avancées

Stash (Sauvegarde Temporaire)

bash

```
# Sauvegarder les modifications en cours
git stash

# Sauvegarder avec un message
git stash save "WIP: travail sur l'authentification"

# Inclure les fichiers non trackés
git stash -u

# Lister tous les stashs
git stash list

# Appliquer le dernier stash
git stash apply

# Appliquer et supprimer le dernier stash
git stash pop

# Appliquer un stash spécifique
git stash apply stash@{2}

# Voir les changements dans un stash
git stash show -p stash@{0}

# Créer une branche depuis un stash
git stash branch nouvelle-branche stash@{0}

# Supprimer un stash
git stash drop stash@{0}

# Supprimer tous les stashs
git stash clear
```

Impact : Sauvegarde temporaire. **Aucune perte** tant que vous ne faites pas `stash drop` ou `stash clear`.

Use Case : Vous travaillez sur une feature, mais devez basculer d'urgence sur un hotfix.

```bash
git stash
git checkout main
git checkout -b hotfix/critical-bug
# Corriger le bug...
git checkout feature/my-work
git stash pop  # Récupérer votre travail
```

Cherry-Pick (Appliquer un Commit Spécifique)

```bash
# Appliquer un commit d'une autre branche
git cherry-pick abc123

# Appliquer plusieurs commits
git cherry-pick abc123 def456

# Appliquer une série de commits
git cherry-pick abc123..xyz789

# Cherry-pick sans commiter automatiquement
git cherry-pick -n abc123
```

Impact : Copie un commit d'une branche vers une autre. Utile pour appliquer un bugfix d'une branche à une autre sans tout merger.

Use Case : Un bugfix a été fait dans `develop`, mais vous devez l'appliquer aussi dans `hotfix/production`.

```bash
git checkout hotfix/production
git cherry-pick abc123  # Le commit du bugfix depuis develop
```

Revert (Annuler un Commit)

```bash
# Créer un nouveau commit qui annule un commit précédent
git revert abc123

# Revert sans créer de commit immédiatement
git revert -n abc123

# Revert un merge commit
git revert -m 1 abc123
```

Impact : Crée un **nouveau commit** qui annule les changements. **N'efface pas l'historique**. Sûr pour les branches publiques.

Reset (Déplacer HEAD)

```bash
```

```
# Soft reset : garde les changements dans le staging area
git reset --soft HEAD~1

# Mixed reset (par défaut) : garde les changements dans le working directory
git reset HEAD~1
git reset --mixed HEAD~1

# Hard reset : SUPPRIME TOUT (⚠️ DANGEREUX)
git reset --hard HEAD~1

# Reset vers un commit spécifique
git reset --hard abc123

# Reset d'un fichier spécifique
git reset HEAD fichier.txt
```

Impact :

- `--soft` : Annule le commit mais garde tout staged. Vous pouvez re-commiter différemment.
- `--mixed` : Annule le commit et unstage. Les fichiers restent modifiés.
- `--hard` : **PERTE DÉFINITIVE** des changements. Retour à l'état du commit.

⚠️ **NE JAMAIS** faire `reset --hard` sur une branche publique partagée !

Rebase Interactif (Réécrire l'Historique)

```
bash

# Modifier les 5 derniers commits
git rebase -i HEAD~5

# Commandes disponibles dans l'éditeur :
# pick   = garder le commit
# reword = modifier le message
# edit   = modifier le contenu du commit
# squash = fusionner avec le commit précédent
# fixup  = fusionner sans garder le message
# drop   = supprimer le commit
```

Impact : Réécrit l'historique. **Ne JAMAIS rebaser** des commits déjà pushés sur une branche publique !

Use Case : Nettoyer l'historique avant de merger une feature branch.

```
bash
```

```
# Avant merge, squasher plusieurs commits "WIP" en un seul
git rebase -i HEAD~10
# Remplacer "pick" par "squash" pour les commits à fusionner
git push --force-with-lease origin feature/my-work
```

Reflog (Historique des Mouvements de HEAD)

```
bash

# Voir tous les mouvements de HEAD
git reflog

# Voir les 20 derniers
git reflog -20

# Récupérer un commit "perdu" après un reset --hard
git reflog
# Trouver le commit voulu (ex: abc123)
git reset --hard abc123

# Créer une branche depuis un état du reflog
git branch branche-recuperee abc123
```

Impact : Sauvegarde locale de TOUS les mouvements de HEAD pendant ~90 jours. Permet de récupérer des commits "perdus".

Bisect (Recherche Dichotomique de Bugs)

```
bash
```

```bash
# Démarrer une recherche de bug
git bisect start

# Marquer le commit actuel comme mauvais
git bisect bad

# Marquer un ancien commit comme bon
git bisect good abc123

# Git va checkout un commit au milieu
# Tester si le bug est présent
git bisect good   # Si pas de bug
git bisect bad    # Si bug présent

# Git continue jusqu'à trouver le commit fautif

# Terminer la recherche
git bisect reset
```

Impact : Recherche automatisée du commit qui a introduit un bug. Très efficace sur de gros historiques.

Tags (Versions)

```bash
```

```
# Créer un tag léger
git tag v1.0.0

# Créer un tag annoté (recommandé)
git tag -a v1.0.0 -m "Version 1.0.0 - Production release"

# Lister tous les tags
git tag

# Voir les détails d'un tag
git show v1.0.0

# Pousser un tag vers GitHub
git push origin v1.0.0

# Pousser tous les tags
git push origin --tags

# Supprimer un tag local
git tag -d v1.0.0

# Supprimer un tag distant
git push origin --delete v1.0.0

# Checkout un tag (mode détaché)
git checkout v1.0.0

# Créer une branche depuis un tag
git checkout -b hotfix-1.0.1 v1.0.0
```

Impact : Marque un point important dans l'historique (releases, versions). Permanent et immuable.

Bonnes Pratiques

1. Commits

✅ **À FAIRE** :

- Commits atomiques (une seule chose à la fois)
- Messages descriptifs et clairs
- Commiter régulièrement (plusieurs fois par jour)
- Utiliser les conventions de nommage (feat, fix, docs...)

❌ **À ÉVITER** :

- Gros commits avec 20 fichiers modifiés
- Messages vagues : "update", "fix stuff", "WIP"
- Commiter du code qui ne compile pas
- Commiter des credentials, API keys, mots de passe

```bash
# Bon commit
git commit -m "feat: ajout validation email avec regex"

# Mauvais commit
git commit -m "update"
```

2. Branches

✅ **À FAIRE** :

- Nommer les branches clairement : `feature/auth-system`, `fix/database-timeout`
- Une branche = une fonctionnalité
- Branches courtes (1-3 jours de travail max)
- Supprimer les branches après merge

❌ **À ÉVITER** :

- Branches qui vivent des semaines
- Noms vagues : `test`, `new-stuff`, `branch1`
- Accumuler des dizaines de branches non mergées

3. Pull Requests

✅ **À FAIRE** :

- Descriptions claires de ce qui a changé
- Lier les issues/tickets concernés
- Demander des reviewers appropriés
- Petites PRs (< 400 lignes de code)
- Répondre rapidement aux commentaires

❌ **À ÉVITER** :

- PRs gigantesques (> 1000 lignes)
- Merger sans review

- Ignorer les commentaires
- PRs avec des commits "WIP"

4. Synchronisation

✅ **À FAIRE** :

- `git pull` AVANT de commencer à travailler
- `git pull` AVANT de pusher
- Pull régulièrement (plusieurs fois par jour)
- Communiquer avec l'équipe

❌ **À ÉVITER** :

- Travailler des jours sans pull
- Forcer push sur des branches partagées
- Ignorer les conflits

5. Sécurité

✅ **À FAIRE** :

- Utiliser `.gitignore` pour exclure :
  - Credentials (`.env`, `config.json`)
  - Dépendances (`node_modules/`, `venv/`)
  - Fichiers de build (`dist/`, `*.pyc`)
  - Fichiers IDE (`.vscode/`, `.idea/`)
- Utiliser SSH plutôt que HTTPS
- Activer 2FA sur GitHub
- Utiliser des tokens avec permissions limitées

❌ **À ÉVITER** :

- Commiter des mots de passe
- Commiter des API keys
- Commiter des certificats SSL
- Commiter des données sensibles

```bash
```

```
# .gitignore typique pour un projet Node.js + Terraform
# Dépendances
node_modules/
package-lock.json

# Variables d'environnement
.env
.env.local
.env.production

# Terraform
*.tfstate
*.tfstate.backup
.terraform/
terraform.tfvars

# IDE
.vscode/
.idea/
*.swp

# OS
.DS_Store
Thumbs.db
```

Commandes de Secours

Annuler des Modifications Non Commitées

```
bash
```

```bash
# Annuler les modifications d'un fichier
git checkout -- fichier.txt
# ou
git restore fichier.txt

# Annuler toutes les modifications
git checkout -- .
# ou
git restore .

# Supprimer les fichiers non trackés
git clean -n  # Voir ce qui serait supprimé
git clean -f  # Supprimer
git clean -fd  # Supprimer fichiers et dossiers
```

Annuler un Commit (Pas Encore Pushé)

```bash
# Garder les modifications
git reset --soft HEAD~1

# Supprimer les modifications (⚠️)
git reset --hard HEAD~1
```

Annuler un Push (⚠️ DANGEREUX)

```bash
# Si personne n'a pullé entre temps
git reset --hard HEAD~1
git push --force-with-lease origin branch-name

# Sinon, utiliser revert (plus sûr)
git revert HEAD
git push origin branch-name
```

Récupérer un Fichier Supprimé

```bash
```

```
# Récupérer depuis le dernier commit
git checkout HEAD -- fichier-supprime.txt

# Récupérer depuis un commit spécifique
git checkout abc123 -- fichier.txt

# Chercher dans l'historique
git log -- fichier-supprime.txt
git checkout abc123 -- fichier-supprime.txt
```

Récupérer après un Reset --hard

bash

```
# Voir l'historique complet de HEAD
git reflog

# Trouver le commit avant le reset
# Exemple : HEAD@{5}: commit: feat: ma fonctionnalité

# Revenir à cet état
git reset --hard HEAD@{5}
```

Changer l'Origine Remote

bash

```
# Voir l'origine actuelle
git remote -v

# Changer l'URL
git remote set-url origin git@github.com:nouveau-user/nouveau-repo.git

# Ajouter un remote supplémentaire
git remote add upstream git@github.com:original/repo.git

# Supprimer un remote
git remote remove upstream
```

Nettoyer les Branches Locales

bash

```
# Voir les branches mergées dans main
git branch --merged main

# Supprimer toutes les branches mergées (sauf main)
git branch --merged main | grep -v "main" | xargs git branch -d

# Supprimer les références aux branches distantes supprimées
git fetch --prune
# ou
git remote prune origin
```

Tableaux de Référence Rapide

Commandes Essentielles Quotidiennes

Commande	Description	Fréquence
`git status`	Voir l'état des fichiers	Constamment
`git add .`	Stager tous les changements	Avant chaque commit
`git commit -m "message"`	Créer un commit	Plusieurs fois par jour
`git pull origin main`	Récupérer les changements	Début de journée, avant push
`git push origin branch`	Envoyer les commits	Après commits
`git checkout -b feature/x`	Créer nouvelle branche	Début de feature
`git log --oneline`	Voir l'historique	Plusieurs fois par jour

Commandes de Branches

Commande	Action	Impact
`git branch`	Lister branches locales	Aucun
`git branch -a`	Lister toutes les branches	Aucun
`git branch nom`	Créer branche	Crée sans basculer
`git checkout -b nom`	Créer et basculer	Bascule vers nouvelle branche
`git switch nom`	Basculer vers branche	Change de branche
`git branch -d nom`	Supprimer branche mergée	Suppression sûre
`git branch -D nom`	Forcer suppression	⚠️ Perte possible
`git push origin --delete nom`	Supprimer branche distante	Suppression sur GitHub

Commandes de Synchronisation

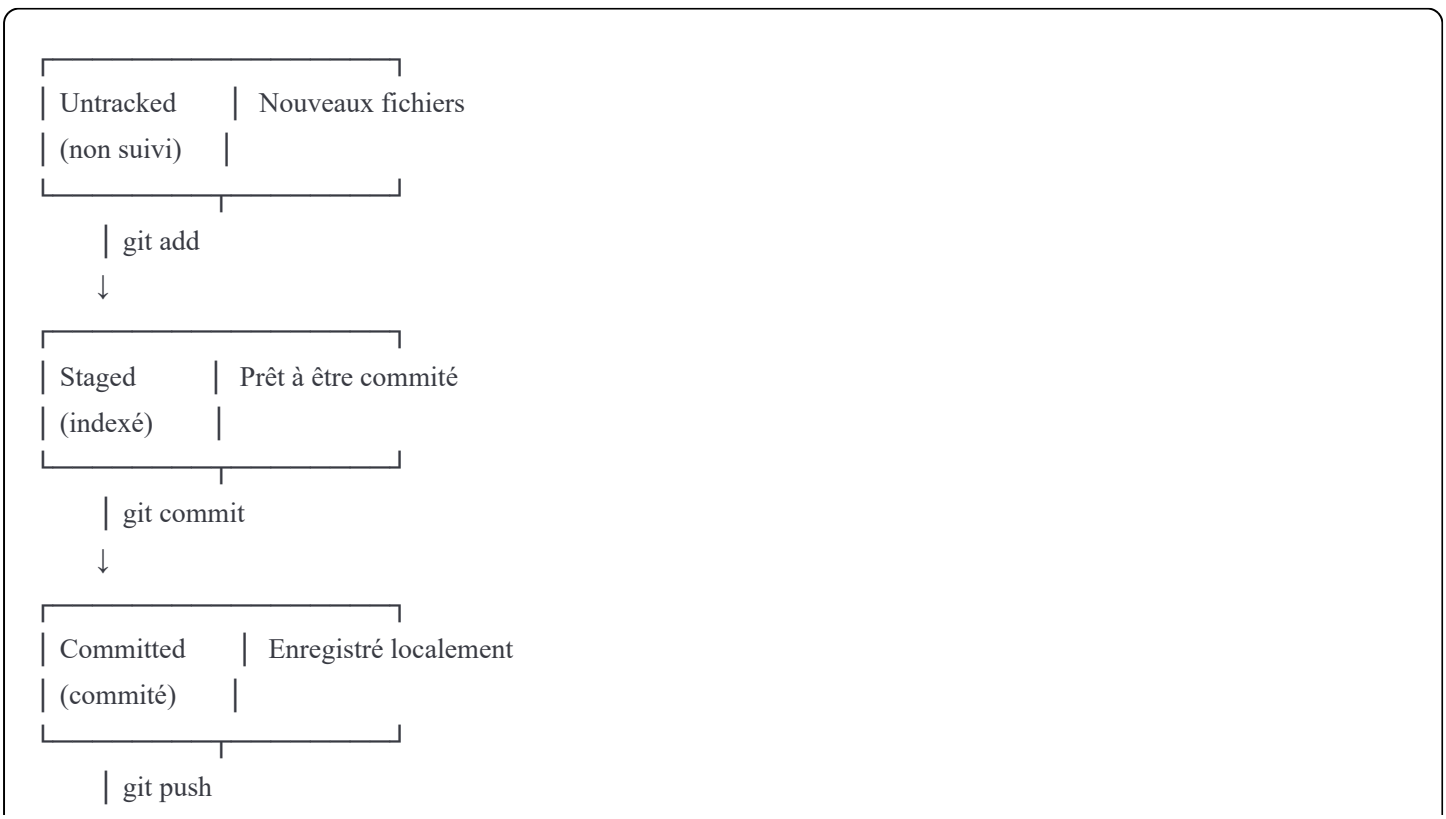
Commande	Action	Impact	Quand Utiliser
`git fetch`	Télécharger sans	Aucun sur code local	Vérifier les changements

Commande	Action fusionner	Impact	Quand Utiliser
`git pull`	Fetch + Merge	Fusionne automatiquement	Mettre à jour branche
`git pull --rebase`	Fetch + Rebase	Historique linéaire	Pour branches propres
`git push`	Envoyer commits	Visible sur GitHub	Après commits
`git push -f`	Force push	⚠️ Écrase historique	Jamais sur branche partagée
`git push --force-with-lease`	Force sécurisé	Écrase si pas de nouveau commit	Après rebase local

Commandes de Modification d'Historique

Commande	Action	Danger	Quand Utiliser
`git commit --amend`	Modifier dernier commit	⚠️ Si déjà pushé	Corriger message/contenu
`git reset --soft HEAD~1`	Annuler commit, garder changements	Faible	Recommiter différemment
`git reset --hard HEAD~1`	Annuler commit et changements	⚠️ ⚠️ Perte définitive	Abandonner travail
`git revert abc123`	Créer commit inverse	Aucun	Annuler sur branche publique
`git rebase -i HEAD~5`	Réécrire historique	⚠️ ⚠️ Si déjà pushé	Nettoyer commits
`git cherry-pick abc123`	Copier un commit	Faible	Appliquer fix spécifique

États des Fichiers Git



```
    ↓
┌──────────────────────────┐
│ Pushed      │ Sur GitHub │
│ (poussé)    │            │
└──────────────────────────┘
```

Résolution de Problèmes Courants

Problème	Solution	Commande
"J'ai modifié le mauvais fichier"	Annuler les modifications	`git checkout -- fichier.txt`
"Je veux annuler mon dernier commit"	Reset soft	`git reset --soft HEAD~1`
"J'ai commit sur la mauvaise branche"	Cherry-pick vers bonne branche	`git cherry-pick abc123`
"Je veux sauvegarder sans commiter"	Utiliser stash	`git stash`
"Conflits lors du pull"	Résoudre puis commit	Éditer → `git add .` → `git commit`
"J'ai fait reset --hard par erreur"	Utiliser reflog	`git reflog` → `git reset --hard HEAD@{n}`
"Ma branche est en retard"	Mettre à jour depuis main	`git rebase main` ou `git merge main`
"Je ne peux pas pusher"	Pull d'abord	`git pull --rebase` puis `git push`

Workflows Complets

Workflow 1 : Nouvelle Fonctionnalité (Feature Branch)

bash

```bash
# Jour 1 - Démarrage
git checkout main
git pull origin main
git checkout -b feature/user-authentication

# Travailler...
git add .
git commit -m "feat: ajout formulaire login"

# Fin de journée
git push -u origin feature/user-authentication

# Jour 2 - Reprise
git pull origin feature/user-authentication  # Au cas où un collègue a contribué
git pull origin main                         # Garder la branche à jour

# Continuer le travail...
git add .
git commit -m "feat: validation JWT tokens"
git push origin feature/user-authentication

# Fin de feature
# Mettre à jour depuis main
git checkout main
git pull origin main
git checkout feature/user-authentication
git rebase main  # Ou merge main si vous préférez

# Résoudre conflits si nécessaire
git push --force-with-lease origin feature/user-authentication

# Sur GitHub : Créer Pull Request
# Après approbation et merge

# Nettoyage
git checkout main
git pull origin main
git branch -d feature/user-authentication
```

Workflow 2 : Correction Urgente (Hotfix)

bash

```bash
# Production a un bug critique !
git checkout main
git pull origin main
git checkout -b hotfix/fix-payment-bug

# Corriger rapidement
git add .
git commit -m "fix: correction calcul TVA dans paiements"

# Push et merge rapide
git push -u origin hotfix/fix-payment-bug

# Sur GitHub : PR rapide → Approbation → Merge

# Déploiement
git checkout main
git pull origin main

# Le fix doit aussi aller dans develop
git checkout develop
git pull origin develop
git merge main  # Ou cherry-pick le commit spécifique
git push origin develop

# Nettoyage
git branch -d hotfix/fix-payment-bug
git push origin --delete hotfix/fix-payment-bug
```

Workflow 3 : Contribution à un Projet Open Source

bash

```bash
# 1. Fork le projet sur GitHub

# 2. Cloner votre fork
git clone git@github.com:votre-user/projet.git
cd projet

# 3. Ajouter le repo original comme upstream
git remote add upstream git@github.com:original-owner/projet.git

# 4. Créer une branche pour votre contribution
git checkout -b feature/add-french-translation

# 5. Travailler sur votre contribution
git add .
git commit -m "feat: ajout traduction française"

# 6. Pousser vers votre fork
git push -u origin feature/add-french-translation

# 7. Sur GitHub : Créer Pull Request vers le repo original

# 8. Si des changements sont demandés
git add .
git commit -m "refactor: amélioration traduction selon feedback"
git push origin feature/add-french-translation

# 9. Garder votre fork à jour avec le repo original
git fetch upstream
git checkout main
git merge upstream/main
git push origin main

# 10. Mettre à jour votre branche feature
git checkout feature/add-french-translation
git rebase main
git push --force-with-lease origin feature/add-french-translation
```

Workflow 4 : Revue de Code d'un Collègue

bash

```bash
# Récupérer la branche du collègue
git fetch origin
git checkout colleague-feature-branch

# Tester localement
npm install  # ou autre commande de setup
npm test

# Si tout est bon, laisser un commentaire approuvant sur GitHub

# Si vous voulez suggérer des changements
git checkout -b colleague-feature-branch-improvements

# Faire vos améliorations
git add .
git commit -m "refactor: optimisation performances"

# Option 1 : Pousser vers une nouvelle branche et créer PR
git push -u origin colleague-feature-branch-improvements

# Option 2 : Pousser directement sur sa branche (si autorisé)
git checkout colleague-feature-branch
git merge colleague-feature-branch-improvements
git push origin colleague-feature-branch
```

Cas d'Usage Avancés

Cas 1 : Récupération Après Catastrophe

Situation : Vous avez fait `git reset --hard` et perdu 3 heures de travail.

```bash
```

```
# 1. Pas de panique ! Vérifier le reflog
git reflog

# Output :
# abc123 HEAD@{0}: reset: moving to HEAD~1
# def456 HEAD@{1}: commit: feat: ma super fonctionnalité (CELUI-CI!)
# xyz789 HEAD@{2}: commit: fix: correction bug

# 2. Récupérer le commit perdu
git reset --hard def456

# Ou créer une branche de secours
git branch recuperation def456
git checkout recuperation

# 3. Vérifier que tout est là
git log
git status

# 4. Continuer normalement
```

Cas 2 : Fusionner Plusieurs Commits en Un Seul

Situation : Vous avez 10 commits "WIP" et voulez les fusionner avant la PR.

bash

```
# 1. Voir vos commits
git log --oneline -10

# 2. Rebase interactif sur les 10 derniers commits
git rebase -i HEAD~10

# 3. Dans l'éditeur qui s'ouvre :
# pick abc123 feat: début authentification
# squash def456 WIP
# squash ghi789 WIP
# squash jkl012 WIP
# pick mno345 feat: ajout validation
# squash pqr678 typo
# ...

# 4. Git ouvrira un éditeur pour le message du commit fusionné
# Éditer le message final : "feat: ajout système authentification complet"

# 5. Forcer le push (branche personnelle uniquement!)
git push --force-with-lease origin feature/auth
```

Cas 3 : Diviser un Gros Commit en Plusieurs Petits

Situation : Vous avez fait un commit avec 15 fichiers et voulez le diviser.

```
bash

# 1. Annuler le dernier commit en gardant les modifications
git reset --soft HEAD~1

# 2. Unstage tous les fichiers
git reset HEAD

# 3. Ajouter et commiter par groupes logiques
git add src/auth/login.js src/auth/register.js
git commit -m "feat: ajout formulaires authentification"

git add src/auth/validation.js
git commit -m "feat: ajout validation emails et mots de passe"

git add tests/auth.test.js
git commit -m "test: ajout tests authentification"

# 4. Pousser
git push origin feature/auth
```

Cas 4 : Appliquer un Fix à Plusieurs Branches

Situation : Un bug critique existe dans `main`, `develop`, et `release-1.0`.

```bash
# 1. Créer le fix sur main
git checkout main
git pull origin main
git checkout -b fix/critical-security-bug

git add .
git commit -m "fix: correction vulnérabilité XSS"
# Notons que c'est le commit abc123

git push origin fix/critical-security-bug
# Merger dans main via PR

# 2. Appliquer sur develop
git checkout develop
git pull origin develop
git cherry-pick abc123
git push origin develop

# 3. Appliquer sur release-1.0
git checkout release-1.0
git pull origin release-1.0
git cherry-pick abc123
git push origin release-1.0

# 4. Vérifier que le fix est partout
git log --all --oneline --grep="vulnérabilité XSS"
```

Cas 5 : Migrer des Commits d'une Branche à l'Autre

Situation : Vous avez commité sur `main` au lieu de votre feature branch.

```bash
```

```
# État actuel : 3 commits sur main qui devraient être sur feature/auth
git log --oneline -5
# abc123 feat: ajout middleware auth
# def456 feat: ajout routes protégées
# ghi789 feat: ajout JWT validation
# jkl012 (ce commit et avant sont OK sur main)

# 1. Créer/basculer vers la bonne branche AVANT ces commits
git checkout -b feature/auth jkl012

# 2. Cherry-pick les commits dans l'ordre
git cherry-pick ghi789
git cherry-pick def456
git cherry-pick abc123

# 3. Pousser la feature branch
git push -u origin feature/auth

# 4. Remettre main au bon endroit
git checkout main
git reset --hard jkl012

# 5. Forcer le push sur main (⚠️ coordonner avec l'équipe!)
git push --force origin main
```

Alias Git Utiles

Ajoutez ces alias à votre `~/.gitconfig` pour gagner du temps :

```bash
# Éditer le fichier de configuration
git config --global --edit

# Ou ajouter directement :
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual 'log --oneline --graph --decorate --all'
```

Fichier ~/.gitconfig complet avec alias utiles :

ini

```
[user]
    name = Votre Nom
    email = votre.email@example.com

[core]
    editor = code --wait
    autocrlf = input

[init]
    defaultBranch = main

[alias]
    # Raccourcis basiques
    st = status -sb
    co = checkout
    br = branch
    ci = commit
    cm = commit -m
    ca = commit -am

    # Logs améliorés
    lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%C
    last = log -1 HEAD --stat
    visual = log --oneline --graph --decorate --all

    # Diffs
    df = diff
    dc = diff --cached

    # Stash
    sl = stash list
    sa = stash apply
    ss = stash save

    # Branches
    branches = branch -a
    remotes = remote -v

    # Annulations
    unstage = reset HEAD --
    undo = reset --soft HEAD~1
    amend = commit --amend --no-edit

    # Nettoyage
    cleanup = !git branch --merged | grep -v '*' | grep -v 'main' | grep -v 'develop' | xargs -n 1 git branch -d
```

```
    # Contribution
    contributors = shortlog -sn

    # Alias complexes
    sync = !git fetch --all && git pull --rebase
    publish = !git push -u origin $(git branch --show-current)
    unpublish = !git push origin --delete $(git branch --show-current)
```

Utilisation :

```bash
# Au lieu de :
git status
git checkout feature-branch
git commit -m "message"

# Vous pouvez faire :
git st
git co feature-branch
git cm "message"

# Log visuel de toutes les branches
git visual

# Voir les contributeurs
git contributors

# Publier la branche actuelle
git publish
```

.gitignore - Templates par Technologie

Node.js / JavaScript

```gitignore
```

```
# Dépendances
node_modules/
npm-debug.log*
yarn-debug.log*
yarn-error.log*
package-lock.json
yarn.lock

# Production
build/
dist/
.cache/

# Environment
.env
.env.local
.env.development.local
.env.test.local
.env.production.local

# Logs
logs/
*.log

# OS
.DS_Store
Thumbs.db

# IDE
.vscode/
.idea/
*.swp
*.swo
*~
```

Python / Django

gitignore

```
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
venv/
env/
ENV/

# Django
*.log
local_settings.py
db.sqlite3
db.sqlite3-journal
media/
staticfiles/

# Tests
.pytest_cache/
.coverage
htmlcov/

# Environment
.env
.venv

# IDE
.vscode/
.idea/
*.swp
```

Terraform

```
gitignore
```

```
# Terraform
*.tfstate
*.tfstate.*
*.tfstate.backup
.terraform/
.terraform.lock.hcl
terraform.tfvars
terraform.tfvars.json
override.tf
override.tf.json
*_override.tf
*_override.tf.json

# Sensitive
*.pem
*.key
secrets.tf

# Logs
crash.log
crash.*.log

# IDE
.vscode/
.idea/
```

React / Next.js

gitignore

```
# Dependencies
node_modules/
.pnp
.pnp.js

# Testing
coverage/

# Next.js
.next/
out/

# Production
build/
dist/

# Misc
.DS_Store
*.pem

# Debug
npm-debug.log*
yarn-debug.log*
yarn-error.log*

# Environment
.env
.env.local
.env.development.local
.env.test.local
.env.production.local

# Vercel
.vercel
```

Checklist Avant Actions Critiques

✅ Avant de Faire un Reset --hard

- [ ] J'ai vérifié que je veux vraiment perdre ces modifications
- [ ] J'ai fait `git stash` si je veux peut-être récupérer plus tard
- [ ] Je suis sur la bonne branche (`git branch`)
- [ ] Je peux retrouver le commit via `git reflog` si besoin

✅ Avant de Faire un Force Push

- [ ] Je suis sur MA branche (pas main, pas develop, pas une branche partagée)
- [ ] J'ai prévenu l'équipe si c'est une branche partagée
- [ ] J'utilise `--force-with-lease` plutôt que `--force`
- [ ] J'ai une sauvegarde (branche locale) au cas où

✅ Avant de Supprimer une Branche

- [ ] La branche a été mergée (`git branch --merged`)
- [ ] Personne d'autre ne travaille dessus
- [ ] La PR a été approuvée et mergée
- [ ] J'ai pull main pour avoir le code fusionné

✅ Avant de Merger une PR

- [ ] Tous les tests passent (CI/CD vert)
- [ ] Au moins 1-2 reviews approuvées
- [ ] Pas de conflits avec la branche cible
- [ ] La branche est à jour avec main/develop
- [ ] La documentation est à jour si nécessaire

✅ Avant de Quitter pour le Weekend

- [ ] Tous mes changements sont commités
- [ ] J'ai pushé mes branches en cours
- [ ] J'ai prévenu l'équipe de mes branches en attente de review
- [ ] Pas de code commenté ou de console.log oubliés
- [ ] Les tests locaux passent

Glossaire des Termes Git

Terme	Définition Simple	Exemple
Commit	Snapshot de votre code à un instant T	Comme sauvegarder un document
Branch	Ligne parallèle de développement	Comme travailler sur une copie
Merge	Fusionner deux branches	Combiner deux versions
Rebase	Rejouer des commits sur une nouvelle base	Réécrire l'historique proprement
HEAD	Où vous êtes actuellement	Le commit actif
Origin	Le dépôt distant (GitHub)	Votre backup cloud
Upstream	Dépôt original (pour les forks)	La source officielle
Fast-forward	Merge sans créer de commit	Historique linéaire
Conflict	Deux versions incompatibles	Git ne sait pas quoi choisir
Stage	Préparer des fichiers pour commit	Zone d'attente
Stash	Mettre de côté temporairement	Tiroir temporaire

Terme	Définition Simple	Exemple
Cherry-pick	Copier un commit	Prendre juste ce qui m'intéresse
Tag	Marquer une version	v1.0.0, v2.0.0
Remote	Dépôt distant	GitHub, GitLab, Bitbucket
Fork	Copie d'un repo sur votre compte	Pour contribuer à l'open source
Clone	Télécharger un repo	Copie locale complète
Pull	Récupérer et fusionner	Mise à jour depuis GitHub
Push	Envoyer vos commits	Upload vers GitHub
Fetch	Récupérer sans fusionner	Juste télécharger les infos

Ressources et Liens Utiles

Documentation Officielle

- **Git Documentation** : https://git-scm.com/doc
- **GitHub Guides** : https://guides.github.com/
- **Atlassian Git Tutorials** : https://www.atlassian.com/git/tutorials

Outils Visuels

- **GitHub Desktop** : Interface graphique pour Git
- **GitKraken** : Client Git visuel avancé
- **SourceTree** : Client Git gratuit par Atlassian
- **VSCode Git Integration** : Extension Git dans VSCode

Sites d'Apprentissage

- **Learn Git Branching** : https://learngitbranching.js.org/ (interactif)
- **Oh Shit, Git!?!** : https://ohshitgit.com/ (solutions aux problèmes courants)
- **Git Explorer** : https://gitexplorer.com/ (trouve la commande dont vous avez besoin)

Commandes Git dans VSCode

- Ctrl+Shift+G : Ouvrir le panneau Git
- Ctrl+Shift+P → "Git: " : Toutes les commandes Git

Notes Finales

Principes d'Or

1. **Commit souvent, push régulièrement** : Petits commits atomiques > gros commits rares
2. **Pull avant de push** : Toujours être à jour avant d'envoyer
3. **Une branche = une fonctionnalité** : Isolation du travail
4. **Communiquer** : Informer l'équipe de vos branches et PRs
5. **Tester avant de merger** : CI/CD et tests locaux doivent passer
6. **Messages clairs** : Commits lisibles = historique utile
7. **Jamais de secrets** : Utiliser .gitignore et .env
8. **Rebase vs Merge** : Rebase pour branches personnelles, merge pour branches publiques
9. **Force push avec précaution** : Seulement sur vos branches
10. **Backup avant actions risquées** : Créer une branche de secours

En Cas de Doute

```bash
# Sauvegarder avant tout
git stash
git branch backup-$(date +%Y%m%d-%H%M%S)

# Vérifier l'état
git status
git log --oneline -10

# Demander de l'aide
# Le reflog est votre ami :)
git reflog
```

Ce guide est un document vivant. Consultez-le régulièrement et adaptez-le à vos besoins !

Version 1.0 - Novembre 2025