Documentation de Référence

Git & GitHub

Du débutant avancé à l'utilisateur expérimenté

Concepts fondamentaux · Commandes · Workflows professionnels · GitHub Actions

---

Version 2025 · Référence complète · Open Source Engineering

Table des matières

Table des matières .... **Erreur ! Signet non défini.**

Partie 1 : Comprendre Git 7

1.1 Histoire et objectifs de Git 7

1.1.1 Pourquoi Git a-t-il été créé ? 7

1.1.2 Les objectifs fondateurs 7

1.1.3 Différences avec les systèmes précédents 7

1.2 Architecture interne de Git 8

1.2.1 Les quatre types d'objets Git 9

1.2.2 Le système de hachage SHA 10

1.2.3 Structure du répertoire .git/ 10

1.2.4 Comment Git stocke les données (snapshots vs diff) 12

1.3 Cycle de vie d'une modification 13

1.3.1 Le Working Directory 13

1.3.2 La Staging Area (zone de préparation) 13

1.3.3 Le dépôt local 14

1.3.4 Le dépôt distant 14

1.4 Les commandes fondamentales 14

1.4.1 git init — Initialiser un dépôt 14

1.4.2 git clone — Cloner un dépôt 14

1.4.3 git status — État du dépôt 15

1.4.4 git add — Stager des modifications 15

1.4.5 git commit — Créer un commit 16

1.4.6 git log — Historique des commits 17

1.4.7 git diff — Voir les différences 18

1.4.8 git show — Inspecter un objet 18

1.4.9 git restore — Restaurer des fichiers 18

1.4.10 git rm et git mv 19

1.4.11 git clean — Nettoyer les fichiers non trackés 19

1.4.12 git stash — Mettre de côté des modifications 19

1.4.13 git tag — Étiqueter des commits 20

1.4.14 git blame — Qui a écrit quoi ? 21

1.4.15 git grep — Chercher dans le code versionné 21
1.4.16 Le fichier .gitignore 22
Partie 2 : Les Branches 23
2.1 Qu'est-ce qu'une branche Git ? 23
2.1.1 HEAD : le pointeur de position 23
2.1.2 Detached HEAD — État déconnecté 24
2.2 Gestion des branches 24
2.2.1 Créer et naviguer entre les branches 24
2.2.2 Stratégies de nommage des branches 25
2.3 git merge — Fusionner des branches 25
2.3.1 Fast-forward merge 25
2.3.2 Merge commit (3-way merge) 26
2.3.3 Résoudre les conflits de merge 26
2.3.4 Options de merge importantes 27
2.4 git rebase — Réécrire l'historique 28
2.4.1 Concept du rebase 28
2.4.2 Rebase interactif — La réécriture de l'historique 29
2.4.3 Merge vs Rebase — Comparaison complète 29
2.5 git cherry-pick — Sélectionner des commits 30
2.6 git reset — Remonter dans le temps 31
2.7 git reflog — Le filet de sécurité ultime 32
Partie 3 : Les Dépôts Distants 34
3.1 Concepts fondamentaux des dépôts distants 34
3.1.1 Remote, Origin et Upstream 34
3.2 Synchronisation : fetch, pull, push 35
3.2.1 git fetch — Télécharger sans intégrer 35
3.2.2 git pull — Télécharger et intégrer 35
3.2.3 git push — Envoyer vers le distant 36
3.2.4 Force push — Comprendre les risques 37
3.2.5 Gérer la divergence 37
Partie 4 : GitHub 39
4.1 Présentation de GitHub 39
4.2 Les Repositories GitHub 39

4.2.1 Types de repositories 39
4.2.2 Paramètres importants d'un repository 40
4.3 Pull Requests — Le cœur de la collaboration 40
4.3.1 Workflow complet d'une Pull Request 40
4.3.2 Les trois stratégies de merge de PR 41
4.3.3 Revue de code — Bonnes pratiques 41
4.4 Issues — Suivi de projet 42
4.4.1 Créer et gérer les Issues 42
4.4.2 Labels, Milestones et templates 42
4.4.3 Références dans les messages de commit 42
4.5 GitHub Pages 42
4.6 Releases — Publier des versions 43
4.6.1 Créer une Release 43
4.6.2 Générer un changelog automatique 43
4.7 GitHub Projects — Gestion de projet 43
4.8 Organisations GitHub 44
4.8.1 Structure d'une organisation 44
4.8.2 Permissions sur les repositories 44
Partie 5 : GitHub Pages 46
5.1 Fonctionnement de GitHub Pages 46
5.2 Types d'URLs GitHub Pages 46
5.3 Configuration de GitHub Pages 46
5.3.1 Via l'interface GitHub 46
5.3.2 Via une branche gh-pages 47
5.4 Domaines personnalisés 47
5.4.1 Configuration DNS 47
5.4.2 HTTPS et certificats SSL 47
5.4.3 Le fichier CNAME 48
5.5 Jekyll — Le générateur de sites intégré 48
5.6 Publier avec GitHub Actions (moderne) 48
5.7 Cas pratiques GitHub Pages 49
5.7.1 Site personnel simple 49
5.7.2 Documentation avec MkDocs 50

5.7.3 Erreurs fréquentes GitHub Pages 51
Partie 6 : GitHub Actions — CI/CD 52
6.1 Concepts fondamentaux 52
6.1.1 CI/CD — Définitions 52
6.1.2 Architecture complète de GitHub Actions 52
6.2 Syntaxe YAML complète 53
6.2.1 Déclencheurs (on:) 53
6.2.2 Contextes et expressions 54
6.2.3 Variables d'environnement et Secrets 54
6.3 Cas pratiques — Workflows complets 55
6.3.1 Workflow Python complet 55
6.3.2 Workflow Node.js complet 56
6.3.3 Workflow Java/Maven 57
6.3.4 Workflow Flutter 58
6.3.5 Fonctionnalités avancées de GitHub Actions 60
6.3.6 Sécurité dans GitHub Actions 61
Partie 7 : Sécurité 62
7.1 Authentification : SSH vs HTTPS 62
7.2 Configuration des clés SSH 62
7.2.1 Générer une paire de clés SSH 62
7.2.2 Configurer ssh-agent 62
7.2.3 Ajouter la clé publique à GitHub 63
7.2.4 Gérer plusieurs comptes GitHub 63
7.3 Personal Access Tokens (PAT) 64
7.3.1 Créer un PAT 64
7.4 Secrets GitHub 64
7.4.1 Repository Secrets 64
7.4.2 Organisation et Environment Secrets 65
7.4.3 Signer les commits GPG 65
Partie 8 : Workflows Professionnels 66
8.1 Git Flow 66
8.1.1 Avantages et inconvénients de Git Flow 67
8.2 GitHub Flow 67

8.3 Trunk Based Development 68
8.3.1 Prérequis pour TBD 69
8.4 Comparaison des trois workflows 69
Partie 9 : Cas Pratiques Complets 70
9.1 Créer un projet personnel de A à Z 70
9.2 Publier un site GitHub Pages complet 71
9.3 Contribuer à un projet Open Source 72
9.4 Travail en équipe — Workflow complet 73
Partie 10 : Référence Rapide 75
10.1 Tableau des commandes Git 75
10.2 Erreurs fréquentes et solutions 76
Erreur : src refspec main does not match any 76
Erreur : non-fast-forward (rejected) 76
Erreur : merge conflict 76
Erreur : detached HEAD 77
Erreur : permission denied (publickey) 77
Erreur : fatal: refusing to merge unrelated histories 77
Erreur : Your local changes would be overwritten by merge 77
10.3 Commandes avancées utiles 78
10.4 Configuration Git recommandée 78
10.5 Glossaire complet 79
Cheat Sheet Git & GitHub 82
⚡ Commandes du quotidien 82
⚡ GitHub Actions YAML minimal 83
⚡ Workflows en 5 lignes 83
⚡ Conventions de commits 83
⚡ Règles d'or 84

Partie 1 : Comprendre Git

1.1 Histoire et objectifs de Git

1.1.1 Pourquoi Git a-t-il été créé ?

En 2005, le noyau Linux utilisait BitKeeper, un système de contrôle de version distribué propriétaire. Suite à un différend de licence entre la communauté Linux et BitMover (l'éditeur de BitKeeper), Linus Torvalds se retrouva sans outil adapté à la gestion du code source du noyau, qui représentait déjà des milliers de contributeurs et des millions de lignes de code.

En quelques semaines seulement, Linus Torvalds créa Git, en s'inspirant des qualités de BitKeeper mais en corrigeant ses défauts. Le nom « Git » est un mot argotique britannique signifiant approximativement « idiot » — Torvalds a déclaré nommer ses projets d'après lui-même : d'abord Linux, puis Git.

> 📅 **Date historique —** Git est né le 7 avril 2005. La version initiale du noyau Linux fut gérée avec Git dès juin 2005 — moins de deux mois après le début du développement.

1.1.2 Les objectifs fondateurs

Linus Torvalds avait des exigences très précises :

- Vitesse — les opérations doivent être quasi-instantanées, même sur de très grands dépôts
- Design simple — architecture minimale mais puissante
- Support fort du développement non-linéaire — branches légères, fusions rapides
- Entièrement distribué — chaque développeur possède une copie complète de l'historique
- Gestion efficace des grands projets — le noyau Linux comme cas de test extrême
- Intégrité des données — chaque objet est identifié par son contenu via SHA

1.1.3 Différences avec les systèmes précédents

Pour comprendre la révolution apportée par Git, il faut comprendre ce qui existait avant.

Les VCS centralisés (CVS, SVN, Perforce)

Dans un système centralisé comme SVN (Subversion), il existe un seul serveur contenant l'historique complet. Les développeurs travaillent sur des copies locales et doivent se connecter au serveur pour valider leurs modifications.

Modèle centralisé vs distribué

```
Architecture SVN (centralisée)
════════════════════════════════

Développeur A ──┐
                │
Développeur B ──┼──► SERVEUR CENTRAL ◄── Historique complet
```

```
                          │
Développeur C ──┘

Problèmes :
• Serveur en panne = travail impossible
• Opérations lentes (réseau requis)
• Branches coûteuses (copie complète)
• Point de défaillance unique
═════════════════════════════════════════

Architecture Git (distribuée)
═════════════════════════════════════════

Dépôt local A ──┐
(historique     │
 complet)       │
                ├──▶ Dépôt distant (GitHub)
Dépôt local B ──┤   (sert de point de
(historique     │    synchronisation)
 complet)       │
                │
Dépôt local C ──┘
(historique complet)

Avantages :
• Travail hors ligne complet
• Opérations locales ultra-rapides
• Pas de point de défaillance unique
• Branches ultra-légères (juste un pointeur)
═════════════════════════════════════════
```

Critère	SVN / CVS	Git
Type	Centralisé	Distribué
Historique	Serveur uniquement	Chaque développeur
Hors ligne	Très limité	Complet
Branches	Coûteuses (copie)	Ultra-légères (pointeur)
Vitesse commit	Lent (réseau)	Instantané (local)
Intégrité	Numéros de révision	Hash SHA cryptographique
Merge	Difficile et lent	Rapide et fréquent

1.2 Architecture interne de Git

Comprendre Git en profondeur nécessite de comprendre comment il stocke réellement les données. Git n'est pas un simple système de diff : c'est une base de données de contenu adressable (content-addressable store), très similaire à un mini-système de fichiers.

1.2.1 Les quatre types d'objets Git

Tout ce que Git stocke est un « objet ». Il n'existe que quatre types d'objets, chacun identifié par un hash SHA-1 (40 caractères hexadécimaux).

① Blob (Binary Large OBject)

Un blob représente le contenu brut d'un fichier. Git ne stocke pas les noms de fichiers dans les blobs — seulement leur contenu. Si deux fichiers ont exactement le même contenu, Git ne les stockera qu'une seule fois (déduplication automatique).

```
# Voir le contenu d'un blob
$ git cat-file -p a1b2c3d4e5f6...
Hello, World!

# Obtenir le type d'un objet
$ git cat-file -t a1b2c3d4e5f6...
blob

# Calculer le hash d'un fichier SANS l'ajouter au dépôt
$ git hash-object fichier.txt
a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4e5f6a1b2
```

② Tree (Arbre)

Un tree représente un répertoire. Il liste les blobs et autres trees qu'il contient, ainsi que leurs noms et permissions. C'est le tree qui fait le lien entre un blob et son nom de fichier.

```
# Afficher le contenu d'un tree
$ git cat-file -p HEAD^{tree}
100644 blob a1b2c3d...  README.md
100644 blob f4e5d6c...  main.py
040000 tree b7c8d9e...  src/
040000 tree 1a2b3c4...  tests/

# Permissions Git :
# 100644 = fichier normal
# 100755 = fichier exécutable
# 120000 = lien symbolique
# 040000 = sous-répertoire (tree)
```

③ Commit

Un commit est un instantané (snapshot) de l'état complet du projet à un moment donné. Il référence un tree (l'état du répertoire racine), un ou plusieurs parents (commits précédents), et contient les métadonnées : auteur, date, message.

```
# Afficher le contenu brut d'un commit
$ git cat-file -p HEAD
tree   b7a8c9d1e2f3a4b5c6d7e8f9a0b1c2d3e4f5a6b7
parent a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
author  Jean Dupont <jean@exemple.com> 1710000000 +0100
committer Jean Dupont <jean@exemple.com> 1710000000 +0100

feat: ajouter la fonctionnalité de connexion
```

```
Implémentation du système d'authentification JWT avec
refresh tokens. Ferme #42.
```

④ Tag annoté

Un tag annoté est un objet à part entière qui pointe vers un commit et ajoute des métadonnées : nom du tagger, date, message de tag, et optionnellement une signature GPG.

```
$ git cat-file -p v1.0.0
object  a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
type    commit
tag     v1.0.0
tagger  Jean Dupont <jean@exemple.com> 1710000000 +0100

Version 1.0.0 — Première version stable
-----BEGIN PGP SIGNATURE-----
...
-----END PGP SIGNATURE-----
```

1.2.2 Le système de hachage SHA

Git utilise SHA-1 (et migre progressivement vers SHA-256) pour identifier chaque objet. Le hash est calculé à partir du type, de la taille et du contenu de l'objet. Cela garantit que deux objets identiques ont le même hash et que toute modification produit un hash complètement différent.

> 🔐 **SHA-1 vs SHA-256 —** La probabilité de collision SHA-1 est astronomiquement faible (environ 1 sur 10^{48}). Dans la pratique, Git est passé à un format SHA-256 pour les nouveaux dépôts afin d'éviter les attaques théoriques contre SHA-1, bien qu'aucune collision pratique n'ait jamais été observée dans un contexte Git.

```
# Comment Git calcule un hash de blob :
$ echo -n "Hello, World!" | git hash-object --stdin
8ab686eafeb1f44702738c8b0f24f2567c36da6d

# Format interne : "blob <taille>\0<contenu>"
$ printf "blob 13\0Hello, World!" | sha1sum
8ab686eafeb1f44702738c8b0f24f2567c36da6d  -
# Les deux sont identiques !
```

1.2.3 Structure du répertoire .git/

Quand vous initialisez un dépôt Git, un répertoire caché .git/ est créé à la racine. C'est là que Git stocke TOUT — l'historique complet, la configuration, les références, etc.

Structure complète de .git/

```
.git/
├── HEAD                    ← Pointeur vers la branche/commit actuel
├── config                  ← Configuration locale du dépôt
├── description             ← Description (GitWeb)
├── COMMIT_EDITMSG          ← Dernier message de commit
├── MERGE_HEAD              ← (présent pendant un merge)
├── MERGE_MSG               ← (présent pendant un merge)
│
├── hooks/                  ← Scripts déclenchés automatiquement
```

```
│    ├── pre-commit         ← Avant chaque commit
│    ├── commit-msg         ← Validation du message
│    ├── pre-push           ← Avant push
│    └── post-merge         ← Après merge
│
├── info/
│    └── exclude            ← .gitignore local (non versionné)
│
├── refs/                   ← Références (branches, tags)
│    ├── heads/             ← Branches locales
│    │    ├── main          ← Fichier contenant le SHA du dernier commit
│    │    └── feature/xyz
│    ├── remotes/           ← Branches distantes (tracking)
│    │    └── origin/
│    │         ├── main
│    │         └── HEAD
│    └── tags/              ← Tags locaux
│         └── v1.0.0
│
├── objects/                ← Base de données des objets
│    ├── pack/              ← Fichiers packés (optimisation)
│    │    ├── pack-xxx.idx
│    │    └── pack-xxx.pack
│    ├── ab/                ← Objets libres (2 premiers hex du SHA)
│    │    └── cd1234...     ← Reste du SHA (38 hex)
│    └── info/
│
└── index                   ← Zone de staging (fichier binaire)
```

Le fichier HEAD

HEAD est un fichier texte qui contient soit une référence symbolique vers une branche (cas normal), soit directement un hash SHA (état « detached HEAD »).

```
# HEAD normal (branche active)
$ cat .git/HEAD
ref: refs/heads/main

# HEAD détaché (detached HEAD)
$ cat .git/HEAD
a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
```

Le répertoire objects/

Les objets sont stockés dans des sous-répertoires nommés par les deux premiers caractères du hash SHA. Ce découpage évite d'avoir trop de fichiers dans un seul répertoire, ce qui serait lent sur certains systèmes de fichiers.

```
# Lister les objets dans objects/
$ find .git/objects -type f | head -10
.git/objects/a1/b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
.git/objects/f4/e5d6c7b8a9b0c1d2e3f4a5b6c7d8e9f0a1b2c3
.git/objects/pack/pack-abc123.idx
.git/objects/pack/pack-abc123.pack

# Inspecter n'importe quel objet
$ git cat-file -p a1b2c3d4
```

```
$ git cat-file -t a1b2c3d4    # type
$ git cat-file -s a1b2c3d4    # taille
```

Le fichier index (zone de staging)

Le fichier .git/index est un fichier binaire qui représente l'état de la zone de staging. Il liste tous les fichiers trackés avec leur mode, hash et timestamp. C'est grâce à ce fichier que git status peut détecter rapidement les modifications.

```
# Inspecter la zone de staging (index)
$ git ls-files --stage
100644 a1b2c3d4e5f6... 0README.md
100644 f4e5d6c7b8a9... 0main.py
100644 b7c8d9e0f1a2... 0src/app.py

# Le 3ème champ (0) est le "stage number"
# 0 = normal
# 1,2,3 = conflits de merge
```

1.2.4 Comment Git stocke les données (snapshots vs diff)

Une idée reçue très répandue : Git stocke des diffs (différences entre versions). C'est FAUX. Git stocke des snapshots (instantanés) complets de chaque version de chaque fichier. Cette approche fondamentale explique pourquoi Git est si rapide.

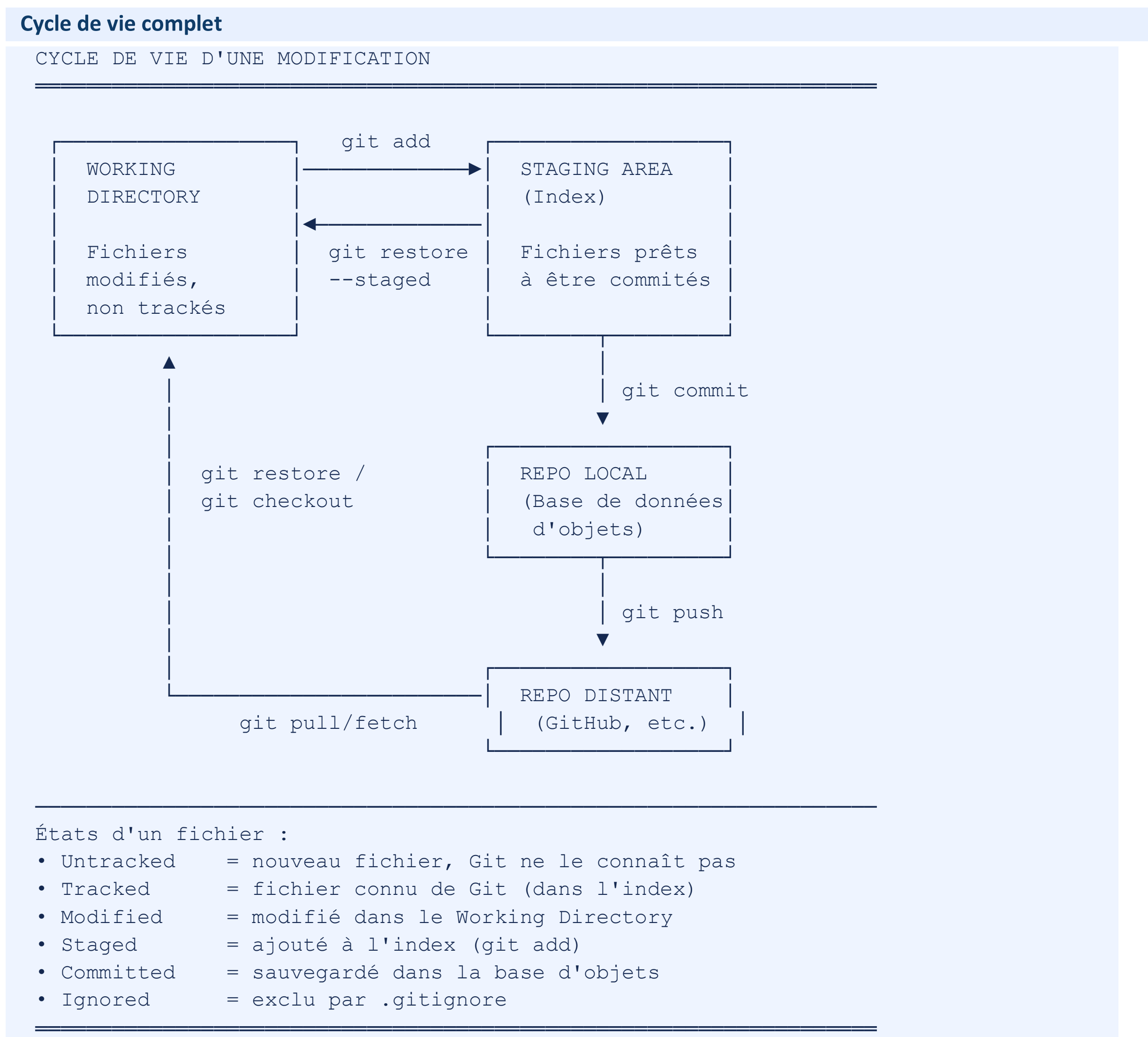
Snapshots vs Diffs

```
Approche diff (SVN) :
════════════════════════════════════════════════
fichier.txt v1 = "Hello"
fichier.txt v2 = v1 + diff("+World")
fichier.txt v3 = v2 + diff("-World +Git")

Pour lire v3, Git doit rejouer tous les diffs → Lent !

────────────────────────────────────────────────

Approche snapshot (Git) :
════════════════════════════════════════════════
Commit A : blob("Hello")         hash=a1b2...
Commit B : blob("Hello World")   hash=f4e5...   (nouveau blob)
Commit C : blob("Hello Git")     hash=b7c8...   (nouveau blob)

Si fichier.txt n'a PAS changé entre B et C :
Commit C pointe vers le MÊME blob que Commit B !
       → Aucune duplication de données.

Pour lire C, Git accède directement au blob → Instantané !
════════════════════════════════════════════════
```

💡 **Pack files et compression —** La déduplication automatique de Git est remarquablement efficace. Git utilise aussi des « pack files » pour regrouper les objets et y appliquer une compression delta (proche des diffs) pour économiser l'espace disque. Ainsi, Git combine le meilleur des deux mondes : accès rapide par snapshot ET stockage compact.

1.3 Cycle de vie d'une modification

Comprendre le cycle de vie d'une modification est essentiel pour maîtriser Git. Une modification traverse plusieurs états avant d'être partagée avec l'équipe.



Cycle de vie complet

1.3.1 Le Working Directory

Le Working Directory (ou répertoire de travail) est simplement le répertoire de votre projet tel que vous le voyez dans votre explorateur de fichiers. C'est ici que vous éditez vos fichiers. Git observe ce répertoire mais n'intervient que sur demande.

1.3.2 La Staging Area (zone de préparation)

La Staging Area (aussi appelée « index ») est une zone intermédiaire unique à Git. Elle vous permet de préparer précisément ce qui fera partie du prochain commit. Vous pouvez par exemple modifier 5 fichiers mais ne stager que 2 d'entre eux pour créer un commit logiquement cohérent.

> 💡 **Pourquoi la Staging Area est précieuse —** La Staging Area est une fonctionnalité qui n'existe pas dans la plupart des autres VCS. Elle peut paraître complexe au début, mais elle est extrêmement puissante : elle vous permet de créer des commits atomiques et logiques même si vous avez travaillé sur plusieurs sujets en même temps.

1.3.3 Le dépôt local

Le dépôt local est la base de données d'objets Git stockée dans .git/objects/. Quand vous faites git commit, Git crée un objet blob pour chaque fichier stagé, un objet tree pour l'arborescence, puis un objet commit qui référence ce tree. Tout cela se passe localement, sans réseau.

1.3.4 Le dépôt distant

Un dépôt distant (remote) est simplement une autre copie du dépôt, généralement hébergée sur un serveur comme GitHub, GitLab ou Bitbucket. git push envoie vos commits locaux vers ce dépôt distant, et git pull les récupère.

1.4 Les commandes fondamentales

1.4.1 git init — Initialiser un dépôt

Crée un nouveau dépôt Git dans le répertoire courant (ou dans un répertoire spécifié).

```
# Initialiser dans le répertoire courant
$ git init

# Initialiser dans un nouveau répertoire
$ git init mon-projet

# Créer un dépôt nu (bare) — pour les serveurs
$ git init --bare mon-projet.git

# Initialiser avec une branche nommée (Git 2.28+)
$ git init -b main
```

> **Dépôts bare —** Un dépôt « bare » (nu) ne contient pas de Working Directory — uniquement la base d'objets et les refs. C'est ce type de dépôt que GitHub crée quand vous créez un nouveau repository. Ne travaillez jamais directement dans un dépôt bare.

1.4.2 git clone — Cloner un dépôt

Copie intégralement un dépôt distant en local : historique complet, branches, tags.

```
# Clone via HTTPS
$ git clone https://github.com/user/repo.git

# Clone via SSH (recommandé pour les contributions)
$ git clone git@github.com:user/repo.git

# Clone dans un répertoire personnalisé
$ git clone https://github.com/user/repo.git mon-dossier
```

```
# Clone peu profond (last N commits seulement)
$ git clone --depth 1 https://github.com/user/repo.git

# Clone une branche spécifique
$ git clone -b develop https://github.com/user/repo.git

# Clone sans télécharger les fichiers (uniquement l'historique)
$ git clone --no-checkout https://github.com/user/repo.git
```

1.4.3 git status — État du dépôt

Affiche l'état de votre Working Directory et de la Staging Area.

```
$ git status

# Sortie typique :
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:       ← Fichiers stagés
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md

Changes not staged for commit: ← Fichiers modifiés non stagés
  (use "git add <file>..." to update what will be committed)
        modified:   main.py

Untracked files:               ← Nouveaux fichiers
  (use "git add <file>..." to include in what will be committed)
        nouveau_fichier.py

# Version courte (-s / --short)
$ git status -s
M  README.md    ← Stagé (colonne gauche)
 M main.py      ← Modifié non stagé (colonne droite)
?? nouveau.py   ← Untracked
```

Format court — git status -s est très pratique au quotidien. La première colonne indique l'état dans la Staging Area, la seconde l'état dans le Working Directory. M = Modified, A = Added, D = Deleted, R = Renamed, ? = Untracked.

1.4.4 git add — Stager des modifications

Ajoute des fichiers ou des modifications à la Staging Area.

```
# Stager un fichier spécifique
$ git add fichier.py

# Stager plusieurs fichiers
$ git add fichier1.py fichier2.py

# Stager tous les fichiers modifiés et nouveaux
$ git add .
$ git add -A            # inclut les suppressions
```

```
# Stager uniquement les modifications (pas les nouveaux fichiers)
$ git add -u

# Stager de manière interactive (choisir des hunks)
$ git add -p            # affiche chaque diff et demande y/n
$ git add --patch

# Stager un répertoire entier
$ git add src/
```

💡 **Bonne pratique : git add -p —** git add -p (ou --patch) est l'une des commandes les plus puissantes et sous-utilisées. Elle vous permet de stager seulement certaines parties d'un fichier (des « hunks »), ce qui vous permet de créer des commits atomiques même si vous avez fait plusieurs modifications distinctes dans le même fichier.

1.4.5 git commit — Créer un commit

Enregistre les modifications stagées dans le dépôt local.

```
# Commit basique (ouvre l'éditeur pour le message)
$ git commit

# Commit avec message en ligne
$ git commit -m "feat: ajouter la page de connexion"

# Stager et commiter en une seule commande (fichiers trackés uniquement)
$ git commit -am "fix: corriger la validation du formulaire"

# Modifier le dernier commit (message ou contenu)
$ git commit --amend
$ git commit --amend -m "Nouveau message"
$ git commit --amend --no-edit   # même message, contenu modifié

# Commit avec co-auteur (bon pour le pair-programming)
$ git commit -m "feat: nouvelle fonctionnalité
>
> Co-authored-by: Marie Martin <marie@exemple.com>"

# Commit vide (utile pour déclencher des CI)
$ git commit --allow-empty -m "chore: trigger CI"
```

Conventions de messages de commit

Les messages de commit sont un aspect crucial de la qualité d'un projet. Le standard le plus répandu est « Conventional Commits ».

Préfixe	Usage	Exemple
`feat`	Nouvelle fonctionnalité	`feat: ajouter authentification JWT`
`fix`	Correction de bug	`fix: corriger crash à la déconnexion`
`docs`	Documentation uniquement	`docs: mettre à jour le README`

style	Formatage, pas de logique	style: reformater selon PEP8
refactor	Refactoring sans bug/feature	refactor: extraire classe UserService
test	Ajout/modification de tests	test: ajouter tests unitaires login
chore	Tâches annexes (deps, config)	chore: mettre à jour dépendances
perf	Amélioration performance	perf: optimiser requête base de données
ci	CI/CD configuration	ci: ajouter workflow GitHub Actions
build	Système de build	build: configurer webpack

📝 **Format Conventional Commits —** Format recommandé : <type>(<scope optionnel>): <description courte en minuscules>\n\n<Corps optionnel>\n\n<Pied de page optionnel (BREAKING CHANGE, Closes #N)>

1.4.6 git log — Historique des commits

Affiche l'historique des commits du dépôt.

```
# Historique basique
$ git log

# Format condensé (une ligne par commit)
$ git log --oneline

# Graphe des branches
$ git log --oneline --graph --all --decorate

# Filtrer par auteur
$ git log --author="Jean Dupont"

# Filtrer par date
$ git log --after="2024-01-01" --before="2024-12-31"

# Filtrer par message
$ git log --grep="feat:"

# Afficher les modifications de chaque commit
$ git log -p

# Afficher les statistiques (fichiers modifiés, insertions/suppressions)
$ git log --stat

# Limiter le nombre de commits
$ git log -10

# Suivre les modifications d'un fichier spécifique
$ git log -- chemin/vers/fichier.py

# Format personnalisé
$ git log --pretty=format:"%h %an %ad %s" --date=short
```

```
# Alias très utile pour visualiser les branches
$ git log --oneline --graph --all --decorate --color
```

Alias git lg — Ajoutez cet alias à votre ~/.gitconfig pour avoir un log visuel pratique :\n[alias]\n   lg = log --oneline --graph --all --decorate\nPuis utilisez simplement : git lg

1.4.7 git diff — Voir les différences

```
# Différences dans le Working Directory (non stagées)
$ git diff

# Différences dans la Staging Area (par rapport au dernier commit)
$ git diff --staged
$ git diff --cached       # synonyme

# Différences entre deux commits
$ git diff abc123 def456

# Différences entre deux branches
$ git diff main develop

# Différences d'un fichier spécifique
$ git diff HEAD -- fichier.py

# Résumé des modifications (sans le diff complet)
$ git diff --stat

# Voir les mots modifiés plutôt que les lignes
$ git diff --word-diff
```

1.4.8 git show — Inspecter un objet

```
# Afficher le dernier commit
$ git show

# Afficher un commit spécifique
$ git show abc123

# Afficher un tag
$ git show v1.0.0

# Afficher un fichier à un commit spécifique
$ git show abc123:chemin/fichier.py

# Afficher seulement les statistiques
$ git show --stat abc123
```

1.4.9 git restore — Restaurer des fichiers

Depuis Git 2.23, git restore remplace les usages de git checkout pour la gestion des fichiers. C'est la commande recommandée pour annuler des modifications.

```
# Annuler les modifications d'un fichier dans le Working Directory
$ git restore fichier.py
$ git restore .               # tous les fichiers
```

```
# Désindexer (unstage) un fichier
$ git restore --staged fichier.py

# Restaurer un fichier à une version spécifique
$ git restore --source=abc123 fichier.py
$ git restore --source=HEAD~2 fichier.py

# Restaurer un fichier depuis une autre branche
$ git restore --source=main fichier.py
```

> ⚠️ **Modifications définitivement perdues —** ATTENTION : git restore fichier.py annule définitivement vos modifications non commitées. Il n'y a pas de 'Annuler' possible. Assurez-vous de vouloir vraiment annuler avant d'exécuter cette commande.

1.4.10 git rm et git mv

```
# Supprimer un fichier et stager la suppression
$ git rm fichier.py

# Supprimer de Git mais garder le fichier localement
$ git rm --cached fichier.py   # utile pour ajouter à .gitignore

# Supprimer un répertoire
$ git rm -r répertoire/

# Renommer/déplacer un fichier
$ git mv ancien_nom.py nouveau_nom.py
$ git mv fichier.py src/fichier.py
```

1.4.11 git clean — Nettoyer les fichiers non trackés

```
# Voir ce qui serait supprimé (dry run)
$ git clean -n
$ git clean --dry-run

# Supprimer les fichiers non trackés
$ git clean -f

# Supprimer fichiers et répertoires non trackés
$ git clean -fd

# Supprimer aussi les fichiers ignorés (.gitignore)
$ git clean -fX   # uniquement ignorés
$ git clean -fx   # ignorés ET non trackés
```

> ⚠️ **Suppression irréversible —** git clean -fd supprime définitivement des fichiers. Toujours lancer -n d'abord pour voir ce qui sera supprimé.

1.4.12 git stash — Mettre de côté des modifications

git stash permet de sauvegarder temporairement des modifications non commitées pour nettoyer votre Working Directory sans perdre votre travail.

```
# Sauvegarder les modifications courantes
$ git stash
$ git stash push -m "WIP: en-cours de refactoring login"

# Inclure les fichiers non trackés
$ git stash -u
$ git stash --include-untracked

# Lister les stashs
$ git stash list
stash@{0}: WIP on main: abc123 Message du commit
stash@{1}: On develop: WIP: en-cours...

# Réappliquer le dernier stash (et le garder dans la liste)
$ git stash apply

# Réappliquer et supprimer le stash
$ git stash pop

# Appliquer un stash spécifique
$ git stash apply stash@{1}

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

# Vider tous les stashs
$ git stash clear

# Voir le contenu d'un stash
$ git stash show -p stash@{0}

# Créer une branche depuis un stash
$ git stash branch ma-branche stash@{0}
```

1.4.13 git tag — Étiqueter des commits

```
# Lister les tags
$ git tag
$ git tag -l "v1.*"   # filtrer

# Créer un tag léger (juste un pointeur)
$ git tag v1.0.0

# Créer un tag annoté (recommandé pour les releases)
$ git tag -a v1.0.0 -m "Version 1.0.0 — Première stable"

# Tag sur un commit spécifique
$ git tag -a v0.9.0 abc123 -m "Pré-release"

# Pousser les tags vers le distant
$ git push origin v1.0.0
$ git push origin --tags   # tous les tags

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

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

# Voir le détail d'un tag
$ git show v1.0.0
```

1.4.14 git blame — Qui a écrit quoi ?

```
# Voir qui a modifié chaque ligne d'un fichier
$ git blame fichier.py

# Résultat :
^a1b2c3d (Jean Dupont 2024-01-15 09:30 +0100  1) def main():
f4e5d6c7 (Marie Martin 2024-02-20 14:15 +0100  2)     print("Hello")
^a1b2c3d (Jean Dupont 2024-01-15 09:30 +0100  3)     return 0

# Ignorer les espaces blancs
$ git blame -w fichier.py

# Limiter à des lignes spécifiques
$ git blame -L 10,20 fichier.py

# Remonter dans l'historique (suivre les copies de code)
$ git blame -C -C fichier.py
```

💡 **Usage moderne —** git blame n'est pas fait pour « blâmer » mais pour comprendre le contexte d'une ligne. Dans la plupart des IDE modernes (VS Code, IntelliJ), cette fonctionnalité est intégrée et affiche les informations directement dans le gutter de l'éditeur.

1.4.15 git grep — Chercher dans le code versionné

```
# Chercher une chaîne dans les fichiers trackés
$ git grep "nom_de_fonction"

# Chercher avec une expression régulière
$ git grep -E "def (get|post)_\w+"

# Afficher le numéro de ligne
$ git grep -n "TODO"

# Chercher dans un commit spécifique
$ git grep "pattern" abc123

# Chercher dans toutes les branches
$ git grep "pattern" $(git branch -a)

# Compter les occurrences
$ git grep -c "import"

# Chercher dans un répertoire spécifique
$ git grep "pattern" -- src/
```

1.4.16 Le fichier .gitignore

Le fichier .gitignore liste les fichiers et répertoires que Git doit ignorer. Il doit être committé dans le dépôt.

```
# Exemple de .gitignore pour un projet Python

# Bytecode Python
__pycache__/
*.py[cod]
*.pyo

# Environnements virtuels
venv/
env/
.env

# IDE
.vscode/
.idea/
*.swp

# Tests et couverture
.coverage
htmlcov/
.pytest_cache/

# Distribution
dist/
build/
*.egg-info/

# Secrets (JAMAIS commiter ces fichiers !)
.env.local
*.pem
*.key
secrets.json

# Système
.DS_Store
Thumbs.db

# Logs
*.log
logs/
```

Générateurs de .gitignore — Il existe un site très pratique pour générer des .gitignore adaptés à votre stack : gitignore.io (ou toptal.com/developers/gitignore). GitHub propose aussi des templates lors de la création d'un repository.

Partie 2 : Les Branches

2.1 Qu'est-ce qu'une branche Git ?

Dans la plupart des VCS, créer une branche est une opération lourde qui copie physiquement les fichiers. Dans Git, une branche est simplement un fichier texte contenant un hash SHA-1 (40 caractères). C'est pourquoi créer une branche est quasi-instantané et ne coûte que 41 octets sur le disque.

Anatomie d'une branche

```
Représentation interne des branches
═══════════════════════════════════════════════════════════════════

.git/refs/heads/main    → contient : "a1b2c3d4..."
.git/refs/heads/feature → contient : "f4e5d6c7..."

───────────────────────────────────────────────────────────────────

Historique Git :

[A] ← [B] ← [C] ← [D]       ← main (pointe vers D)
                  ↑
                  └── [E] ← [F]  ← feature (pointe vers F)

HEAD → main (on est sur la branche main)

───────────────────────────────────────────────────────────────────

Une branche = un pointeur mobile vers un commit.
Quand on crée un nouveau commit, le pointeur avance automatiquement.
═══════════════════════════════════════════════════════════════════
```

2.1.1 HEAD : le pointeur de position

HEAD est un pointeur spécial qui indique « où vous êtes actuellement ». Normalement, HEAD pointe vers une branche (HEAD → main → commit D). Quand vous commitez, la branche avance et HEAD suit automatiquement.

```
# Voir où pointe HEAD
$ cat .git/HEAD
ref: refs/heads/main

# Équivalent via git
$ git symbolic-ref HEAD
refs/heads/main

# Commit vers lequel pointe HEAD
$ git rev-parse HEAD
a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0
```

2.1.2 Detached HEAD — État déconnecté

Un « Detached HEAD » signifie que HEAD pointe directement vers un commit plutôt que vers une branche. C'est un état légitime mais potentiellement dangereux si vous n'en avez pas conscience.

```
# Entrer en detached HEAD (checkout vers un commit)
$ git checkout abc123

# Git affiche un avertissement :
# HEAD is now at abc123 Votre commit
# You are in 'detached HEAD' state. You can look around, make experimental
# changes and commit them, and you can discard any commits you make in this
# state without impacting any branches by switching back to a branch.

# Si vous commitez en detached HEAD, ces commits ne sont rattachés à aucune
branche.
# Ils seront nettoyés par le garbage collector de Git après un moment !

# Sauver un detached HEAD en créant une branche
$ git branch nouvelle-branche    # créer une branche sur ce commit
$ git switch nouvelle-branche    # y aller
```

⚠️ **Detached HEAD : ne pas paniquer —** Si vous vous retrouvez accidentellement en detached HEAD, ne paniquez pas. Vos commits ne sont pas perdus immédiatement. Créez simplement une branche avec git branch ma-sauvegarde avant de partir, et vos commits seront préservés.

2.2 Gestion des branches

2.2.1 Créer et naviguer entre les branches

```
# Lister toutes les branches locales
$ git branch
* main
  develop
  feature/login

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

# Lister les branches distantes
$ git branch -r

# Créer une nouvelle branche (sans y aller)
$ git branch feature/nouvel-element

# Créer et aller sur une nouvelle branche (Git 2.23+)
$ git switch -c feature/nouvel-element

# Ancienne syntaxe (toujours valide)
$ git checkout -b feature/nouvel-element

# Changer de branche
$ git switch main
```

```
$ git checkout main    # ancienne syntaxe

# Créer une branche à partir d'un commit spécifique
$ git branch hotfix/bug-123 abc123

# Renommer la branche courante
$ git branch -m nouveau-nom

# Renommer une branche spécifique
$ git branch -m ancien-nom nouveau-nom

# Supprimer une branche (seulement si mergée)
$ git branch -d feature/login

# Forcer la suppression (même non mergée)
$ git branch -D feature/experimente
```

2.2.2 Stratégies de nommage des branches

Convention	Usage	Exemple
feature/<nom>	Nouvelle fonctionnalité	feature/user-authentication
fix/<nom>	Correction de bug	fix/null-pointer-login
hotfix/<nom>	Correction urgente en production	hotfix/security-patch-2024
release/<ver>	Préparation d'une release	release/v2.1.0
docs/<nom>	Documentation	docs/api-reference-update
chore/<nom>	Tâche technique	chore/update-dependencies
refactor/<nom>	Refactoring	refactor/extract-user-service
test/<nom>	Tests	test/integration-tests-auth

2.3 git merge — Fusionner des branches

Le merge (fusion) intègre les modifications d'une branche dans une autre. Il existe deux stratégies principales : fast-forward et merge commit.

2.3.1 Fast-forward merge

Un fast-forward est possible quand la branche cible n'a pas avancé depuis le point de divergence. Dans ce cas, Git déplace simplement le pointeur de la branche cible vers le dernier commit de la branche source. Il n'y a pas de commit de merge créé.

Fast-forward merge

```
Avant le merge (fast-forward possible) :

main → [A] ← [B] ← [C]
                    ↑
```

```
                    feature ← [D] ← [E]

HEAD → main
main n'a pas avancé depuis [C] (point de divergence de feature)

─────────────────────────────────────────────

Après : git switch main && git merge feature

[A] ← [B] ← [C] ← [D] ← [E]
                            ↑
                       main, feature

main a simplement avancé vers [E] — pas de commit de merge !
```

```
# Fast-forward merge
$ git switch main
$ git merge feature/login

# Forcer un merge commit même si FF est possible
$ git merge --no-ff feature/login

# Voir si un merge sera FF ou non
$ git merge --dry-run feature/login 2>/dev/null; echo "FF possible"
$ git merge-base main feature/login
```

2.3.2 Merge commit (3-way merge)

Quand les deux branches ont avancé indépendamment, Git doit créer un commit de merge spécial qui a deux parents. Git trouve l'ancêtre commun et effectue une fusion à 3 voies (3-way merge).

Merge commit (3-way merge)

```
Avant le merge (merge commit nécessaire) :

        [D] ← [E]    ← feature
       /
[A] ← [B] ← [C]     ← main

Les deux branches ont avancé depuis [B]

─────────────────────────────────────────────

Après : git merge feature

        [D] ← [E] ──┐
       /              ↓
[A] ← [B] ← [C] ← [M]  ← main
                   (merge commit : 2 parents)

[M] = merge commit créé automatiquement
Message : "Merge branch 'feature' into main"
```

2.3.3 Résoudre les conflits de merge

Un conflit se produit quand les deux branches ont modifié la même partie d'un fichier. Git ne peut pas décider automatiquement quelle version garder.

```
# Déclencher un merge qui produit des conflits
$ git merge feature/login
Auto-merging auth.py
CONFLICT (content): Merge conflict in auth.py
Automatic merge failed; fix conflicts and then commit the result.

# Voir les fichiers en conflit
$ git status
both modified:   auth.py

# Contenu du fichier en conflit :
<<<<<<< HEAD           ← début de votre version (main)
def login(user, pwd):
    return db.verify(user, pwd)
=======                ← séparateur
def login(username, password, remember=False):
    return auth.verify(username, password, remember)
>>>>>>> feature/login  ← fin de la version entrante

# Résoudre manuellement (éditer le fichier)
# Puis stager la résolution
$ git add auth.py

# Finaliser le merge
$ git commit

# Ou annuler le merge complètement
$ git merge --abort
```

Outils de résolution de conflits — Les outils de mergetool peuvent grandement simplifier la résolution de conflits. Configurez votre outil préféré avec : git config --global merge.tool vscode (ou vimdiff, meld, kdiff3, etc.). Lancez-le avec : git mergetool

Marqueur	Signification
`<<<<<<< HEAD`	Début de votre version (branche courante)
`=======`	Séparateur entre les deux versions
`>>>>>>> branche`	Fin de la version entrante (branche mergée)
`||||||| base`	Version de base (ancêtre commun) — avec merge.conflictstyle=diff3

2.3.4 Options de merge importantes

```
# Squash merge (condense tous les commits en un seul)
$ git merge --squash feature/login
$ git commit -m "feat: ajouter authentification"  # commit manuel nécessaire

# Merge sans commit (staging uniquement)
$ git merge --no-commit feature/login

# Stratégie ours (ignorer complètement les modifications de l'autre branche)
$ git merge -s ours branche-obsolete
```

```
# Résoudre automatiquement les conflits (préférer notre version)
$ git merge -X ours feature/login

# Préférer la version entrante
$ git merge -X theirs feature/login
```

2.4 git rebase — Réécrire l'historique

Le rebase est l'une des fonctionnalités les plus puissantes et les plus incomprises de Git. Il permet de « replanter » une branche sur un autre commit, en rejouant ses commits un par un.

2.4.1 Concept du rebase

Rebase : concept

```
Situation initiale :

       [D] ← [E]   ← feature (4 commits d'avance sur main)
      /
[A] ← [B] ← [C]    ← main (2 commits d'avance sur feature)

─────────────────────────────────────────────────────

git switch feature
git rebase main

─────────────────────────────────────────────────────

Après le rebase :

                   [D'] ← [E']   ← feature
                   /
[A] ← [B] ← [C]
             ↑
            main

[D'] et [E'] sont des NOUVEAUX commits avec de nouveaux hashes.
Ils ont le même contenu que [D] et [E] mais des parents différents.
Ils sont "replantés" sur [C] au lieu de [B].

Historique résultant : linéaire et propre !
[A] ← [B] ← [C] ← [D'] ← [E']
```

```
# Rebase de la branche courante sur main
$ git switch feature
$ git rebase main

# Rebase interactif (réécriture avancée de l'historique)
$ git rebase -i HEAD~3     # réécrire les 3 derniers commits
$ git rebase -i main       # tout ce qui n'est pas dans main

# Continuer après résolution de conflit
$ git rebase --continue
```

```
# Passer le commit courant (en cas de conflit)
$ git rebase --skip

# Annuler le rebase
$ git rebase --abort

# Rebase sur un commit spécifique
$ git rebase abc123
```

2.4.2 Rebase interactif — La réécriture de l'historique

Le rebase interactif (git rebase -i) est l'outil le plus puissant pour nettoyer un historique avant de partager son travail.

```
$ git rebase -i HEAD~4

# L'éditeur s'ouvre avec :
pick a1b2c3d fix: typo dans README
pick f4e5d6c feat: début de la page connexion
pick b7c8d9e feat: formulaire connexion
pick 1a2b3c4 wip: a nettoyer

# Commandes disponibles (remplacer 'pick') :
# pick (p)   = utiliser le commit tel quel
# reword (r) = modifier le message du commit
# edit (e)   = modifier le contenu du commit
# squash (s) = fusionner avec le commit précédent (garder message)
# fixup (f)  = fusionner avec le commit précédent (ignorer message)
# drop (d)   = supprimer ce commit
# exec (x)   = exécuter une commande shell

# Exemple : nettoyer l'historique
pick a1b2c3d fix: typo dans README
pick f4e5d6c feat: début de la page connexion
squash b7c8d9e feat: formulaire connexion
drop 1a2b3c4 wip: a nettoyer
# Résultat : 2 commits propres au lieu de 4
```

💡 **Bonne pratique : rebase avant PR —** Le rebase interactif est idéal pour préparer une Pull Request. Avant de la soumettre, vous pouvez squasher les commits de travail, reformuler les messages, et présenter un historique clair et logique à vos reviewers.

2.4.3 Merge vs Rebase — Comparaison complète

Critère	Merge	Rebase
Historique	Non-linéaire (commit de merge)	Linéaire (propre)
Commits originaux	Préservés avec leurs hashes	Recréés avec nouveaux hashes
Conflits	Résolus une fois	Résolus pour chaque commit rejoué
Auditabilité	Montre l'historique réel	Montre un historique idéalisé
Sécurité distante	Toujours sûr	⚠️ Dangereux si déjà pushé

Complexité	Simple et automatique	Requiert compréhension Git
Recommandé pour	Branches longues, travail équipe	Branches courtes, nettoyage local
Commande principale	`git merge feature`	`git rebase main`

🔴 **Règle d'or : ne jamais rebase des commits pushés publics —** LA RÈGLE D'OR DU REBASE : Ne jamais rebase des commits qui ont déjà été poussés vers un dépôt partagé (sauf si vous êtes le seul utilisateur de cette branche). Rebase réécrit les commits → nouveaux hashes → l'historique des autres développeurs diverge → chaos garanti.

2.5 git cherry-pick — Sélectionner des commits

git cherry-pick applique les modifications d'un commit spécifique sur la branche courante, sans merger toute la branche. Très utile pour appliquer un hotfix sur plusieurs branches.

Cherry-pick : sélectionner un commit

```
Situation : un fix important est sur 'develop', on veut l'appliquer sur 'main'

develop: [A] ← [B] ← [fix] ← [C] ← [D]
main:    [A] ← [B] ← [E] ← [F]

git switch main
git cherry-pick <hash-du-fix>

Résultat :
develop: [A] ← [B] ← [fix] ← [C] ← [D]
main:    [A] ← [B] ← [E] ← [F] ← [fix']
                                    (nouveau hash, même contenu)
```

```
# Appliquer un commit spécifique
$ git cherry-pick abc123

# Appliquer plusieurs commits
$ git cherry-pick abc123 def456 ghi789

# Appliquer une plage de commits (excluant le premier)
$ git cherry-pick abc123..def456

# Appliquer sans commiter (juste stager)
$ git cherry-pick -n abc123
$ git cherry-pick --no-commit abc123

# Cherry-pick avec modification du message
$ git cherry-pick -e abc123

# Continuer après résolution de conflit
$ git cherry-pick --continue

# Annuler
$ git cherry-pick --abort
```

2.6 git reset — Remonter dans le temps

git reset permet de déplacer HEAD (et la branche courante) vers un commit précédent. Il existe trois modes qui ont des effets différents sur la Staging Area et le Working Directory.

git reset --soft vs --mixed vs --hard

```
SCHÉMA DES TROIS MODES DE RESET
═══════════════════════════════════════════════════════════════════════════════

État initial :

HEAD → main → [commit C]

Working Directory : modifié
Staging Area      : stagé
Repo local        : commit C

────────────────────────────────────────────────────────────────────────────

git reset --soft HEAD~1

HEAD → main → [commit B]  ← repositionné

Working Directory : inchangé ✓
Staging Area      : contient les modifications de C ✓
Repo local        : B (C est "perdu")

Utilisation : corriger un commit oublié (réajouter un fichier)

────────────────────────────────────────────────────────────────────────────

git reset --mixed HEAD~1  (défaut)

HEAD → main → [commit B]

Working Directory : inchangé ✓
Staging Area      : vidée, modifications de C dans WD
Repo local        : B

Utilisation : décommiter pour restructurer le staging

────────────────────────────────────────────────────────────────────────────

git reset --hard HEAD~1

HEAD → main → [commit B]

Working Directory : ⚠ RÉINITIALISÉ comme commit B
Staging Area      : ⚠ VIDÉE
Repo local        : B

Utilisation : annuler complètement (IRRÉVERSIBLE sans reflog)
═══════════════════════════════════════════════════════════════════════════════
```

```
# Remonter d'un commit (le contenu reste stagé)
$ git reset --soft HEAD~1
```

```
# Remonter d'un commit (le contenu revient dans WD, non stagé)
$ git reset HEAD~1
$ git reset --mixed HEAD~1    # identique

# Annuler complètement les N derniers commits
$ git reset --hard HEAD~3

# Revenir à un commit spécifique
$ git reset --hard abc123

# Différence entre reset et revert :
# reset : réécrit l'historique (dangereux si pushé)
# revert : crée un NOUVEAU commit qui annule (safe)

# Créer un commit d'annulation (safe pour historique partagé)
$ git revert HEAD         # annuler le dernier commit
$ git revert abc123       # annuler un commit spécifique
$ git revert HEAD~3..HEAD # annuler plusieurs commits
```

🔴 **reset vs revert sur branches partagées —** Préférez git revert sur les branches partagées. git reset réécrit l'historique et peut causer des problèmes pour vos collègues. git revert ajoute un commit d'annulation sans modifier l'historique existant.

2.7 git reflog — Le filet de sécurité ultime

Le reflog (reference log) est un journal local qui enregistre TOUS les déplacements de HEAD, même les commits « perdus » après un reset --hard ou un mauvais rebase. C'est le filet de sécurité ultime de Git.

```
# Afficher le reflog
$ git reflog

# Résultat typique :
abc123 HEAD@{0}: commit: feat: nouvelle page
f4e5d6 HEAD@{1}: reset: moving to HEAD~1
b7c8d9 HEAD@{2}: commit: feat: ajout formulaire
1a2b3c HEAD@{3}: checkout: moving from main to feature
...

# Récupérer un commit "perdu" après reset --hard
$ git reflog                        # trouver le hash
$ git checkout abc123               # inspecter
$ git branch sauvegarde abc123      # créer une branche dessus

# Récupérer une branche supprimée
$ git reflog | grep "la-branche"
$ git checkout -b la-branche abc123

# Annuler un mauvais reset --hard
$ git reflog
$ git reset --hard HEAD@{2}         # revenir 2 positions en arrière

# Durée de rétention du reflog
# Par défaut : 90 jours (commits accessibles)
```

```
# Commits inaccessibles : 30 jours
$ git config gc.reflogExpire 180.days.ago
```

⚠️ **Le reflog est local uniquement —** Le reflog est UNIQUEMENT LOCAL. Il n'est pas pushé vers le distant et ne fait pas partie de l'historique partagé. Si vous réinstallez Git ou clonez un nouveau repo, le reflog sera vide. C'est pourquoi il faut agir vite après une manipulation malheureuse.

🆘 **Procédure d'urgence après reset --hard —** Si vous avez fait un reset --hard accidentel, le reflog peut vous sauver. La procédure d'urgence : 1) git reflog pour voir l'historique, 2) identifier le hash du commit voulu, 3) git reset --hard <hash> pour revenir, 4) git branch sauvegarde si vous voulez préserver l'état actuel avant.

Partie 3 : Les Dépôts Distants

3.1 Concepts fondamentaux des dépôts distants

3.1.1 Remote, Origin et Upstream

Ces trois termes sont souvent confondus. Voici leurs définitions précises :

Terme	Définition	Contexte d'utilisation
remote	Tout dépôt Git distant (une URL + un alias)	Terme générique pour n'importe quel dépôt distant
origin	Alias conventionnel du dépôt depuis lequel vous avez cloné	Créé automatiquement par git clone
upstream	Alias conventionnel du dépôt original d'un fork	Utilisé quand on travaille sur un fork

Remote / Origin / Upstream

```
Contexte fork (contribution open source) :
══════════════════════════════════════════════════════════

microsoft/vscode           ← upstream (dépôt original)
      |
      | fork (via GitHub)
      ↓
vous/vscode                ← origin (votre fork sur GitHub)
      |
      | git clone
      ↓
~/code/vscode              ← dépôt local

remote -v :
origin   git@github.com:vous/vscode.git (fetch/push)
upstream git@github.com:microsoft/vscode.git (fetch)
══════════════════════════════════════════════════════════
```

```
# Lister tous les remotes
$ git remote
origin
upstream

# Lister avec les URLs
$ git remote -v
origin   git@github.com:vous/projet.git (fetch)
origin   git@github.com:vous/projet.git (push)
upstream git@github.com:original/projet.git (fetch)
upstream git@github.com:original/projet.git (push)
```

```
# Ajouter un remote
$ git remote add upstream https://github.com/original/projet.git

# Modifier l'URL d'un remote
$ git remote set-url origin git@github.com:vous/projet.git

# Supprimer un remote
$ git remote remove upstream
$ git remote rm upstream

# Inspecter un remote (branches, HEAD)
$ git remote show origin
```

3.2 Synchronisation : fetch, pull, push

3.2.1 git fetch — Télécharger sans intégrer

git fetch télécharge les données du dépôt distant (commits, branches, tags) SANS modifier votre Working Directory ni votre branche courante. Il met à jour les « tracking branches » (origin/main, etc.) mais laisse votre travail local intact.

```
# Télécharger tous les changements d'origin
$ git fetch origin
$ git fetch              # équivalent si origin est le remote par défaut

# Télécharger tous les remotes
$ git fetch --all

# Télécharger une branche spécifique
$ git fetch origin main

# Télécharger et nettoyer les branches distantes supprimées
$ git fetch --prune
$ git fetch -p

# Après fetch, voir la différence
$ git log main..origin/main --oneline   # commits à intégrer
$ git diff main origin/main             # différences
```

> 💡 **Toujours préférer fetch avant pull —** git fetch est la commande la plus sûre pour se synchroniser avec le distant. Elle n'a aucun effet sur votre travail local. C'est une bonne pratique de faire git fetch régulièrement pour voir ce que font vos collègues sans risquer de perturber votre travail.

3.2.2 git pull — Télécharger et intégrer

git pull est équivalent à git fetch suivi de git merge (par défaut). Il télécharge les modifications ET les intègre directement dans votre branche courante.

```
# Pull standard (fetch + merge)
$ git pull

# Pull depuis un remote et branche spécifiques
```

```
$ git pull origin main

# Pull avec rebase au lieu de merge (historique plus propre)
$ git pull --rebase
$ git pull -r

# Configurer le comportement par défaut
$ git config pull.rebase true    # toujours utiliser rebase
$ git config pull.ff only        # uniquement fast-forward (le plus sûr)

# Pull avec fast-forward uniquement (échoue si FF impossible)
$ git pull --ff-only
```

fetch vs pull

```
Différence fetch vs pull :
══════════════════════════════════════════════════════════

git fetch origin
─────────────────────────────────────────────────────────
Avant : main → [A] ← [B]      origin/main → [A] ← [B]
Distant: main = [A] ← [B] ← [C] ← [D]

Après fetch :
Local  : main → [A] ← [B]       (inchangé)
Tracking: origin/main → [D]     (mis à jour)

Vous pouvez inspecter sans risque, puis décider d'intégrer.

══════════════════════════════════════════════════════════

git pull (= fetch + merge)
─────────────────────────────────────────────────────────
Avant : main → [A] ← [B]
Distant: main = [A] ← [B] ← [C] ← [D]

Après pull :
Local  : main → [D]  (intégré directement !)

Plus rapide mais moins de contrôle.
══════════════════════════════════════════════════════════
```

3.2.3 git push — Envoyer vers le distant

```
# Push vers le remote par défaut (origin)
$ git push

# Push une branche spécifique
$ git push origin main
$ git push origin feature/login

# Push et créer le tracking (--set-upstream)
$ git push -u origin feature/login
# Après : git push suffira (sans préciser origin feature/login)

# Push tous les tags
$ git push --tags
```

```
# Push un tag spécifique
$ git push origin v1.0.0

# Supprimer une branche distante
$ git push origin --delete feature/login
$ git push origin :feature/login    # syntaxe alternative

# Push avec force (dangereux !)
$ git push --force
$ git push -f

# Push avec force mais sécurisé
$ git push --force-with-lease       # recommandé
```

3.2.4 Force push — Comprendre les risques

🔴 **Utiliser --force-with-lease jamais --force —** --force-with-lease est presque toujours préférable à --force. Il vérifie que la version distante correspond à ce que vous avez fetchée localement. Si quelqu'un a pushé entre-temps, le push échoue (vous évitant d'écraser son travail).

```
# Scénario : vous avez rebasé votre branche (nouveaux hashes)
# L'historique local diverge du distant

# ❌ Dangereux : peut écraser le travail d'un collègue
$ git push --force

# ☑ Plus sûr : vérifie que rien n'a changé côté distant
$ git push --force-with-lease

# Message en cas d'échec :
# To github.com:user/repo.git
# ! [rejected] feature/login -> feature/login
#   (stale info) hint: Updates were rejected because the tip of your
#   current branch is behind its remote counterpart.

# Quand est-ce légitime de force-pusher ?
# • Après un git rebase -i sur votre propre branche de feature
# • Pour corriger un commit accidentel avec des données sensibles
# • Sur une branche que VOUS SEUL utilisez
```

3.2.5 Gérer la divergence

```
# Situation : votre branche locale a divergé du distant
$ git push
# To github.com:user/repo.git
# ! [rejected] main -> main (non-fast-forward)
# error: failed to push some refs to 'github.com:user/repo.git'
# hint: Updates were rejected because the tip of your current
#       branch is behind its remote counterpart.

# Solution 1 : Récupérer et merger
$ git pull
$ git push
```

```
# Solution 2 : Récupérer et rebase (historique linéaire)
$ git pull --rebase
$ git push

# Solution 3 : Voir exactement ce qui diverge
$ git fetch
$ git log --oneline HEAD..origin/main   # commits distants non locaux
$ git log --oneline origin/main..HEAD   # commits locaux non distants
```

Partie 4 : GitHub

4.1 Présentation de GitHub

GitHub est une plateforme web d'hébergement de dépôts Git, fondée en 2008 et acquise par Microsoft en 2018. Ce n'est PAS Git : GitHub est un service qui utilise Git comme système de versioning, et y ajoute une couche de collaboration, de gestion de projet et d'automatisation.

Git	GitHub
Outil en ligne de commande local	Service web hébergé
Créé par Linus Torvalds (2005)	Fondé en 2008, acquis par Microsoft (2018)
Open source (licence LGPL)	Propriétaire (certains outils open source)
Gère les versions de fichiers	Héberge, visualise, collabore sur Git
Fonctionne 100% hors ligne	Nécessite une connexion internet
Aucune notion d'utilisateur/équipe	Gestion d'équipes, permissions, organisations

4.2 Les Repositories GitHub

4.2.1 Types de repositories

Type	Visibilité	Usage
Public	Tout le monde peut lire	Open source, portfolio, documentation
Privé	Uniquement les collaborateurs	Projets personnels, code propriétaire
Fork	Copie d'un repo public/privé	Contribuer à un projet externe
Archivé	Lecture seule pour tous	Projet terminé, plus maintenu

```
# Créer un repo depuis la CLI (GitHub CLI)
$ gh repo create mon-projet --public
$ gh repo create mon-projet --private
$ gh repo clone user/repo

# Via l'API GitHub
$ curl -H "Authorization: token ghp_xxx" \
  -d '{"name":"mon-projet","private":false}' \
  https://api.github.com/user/repos
```

4.2.2 Paramètres importants d'un repository

- Branch protection rules — Protéger main contre les pushs directs
- CODEOWNERS — Définir les reviewers automatiques par répertoire
- Topics — Tags pour la découverte
- README.md — Page d'accueil du projet
- LICENSE — Définir les droits d'utilisation
- CONTRIBUTING.md — Guide pour les contributeurs
- .github/ — Templates pour Issues et PRs, workflows Actions

4.3 Pull Requests — Le cœur de la collaboration

Une Pull Request (PR) est une demande d'intégration de modifications d'une branche dans une autre. C'est l'outil central de la collaboration sur GitHub : elle permet la revue de code, les discussions, et l'intégration contrôlée.

4.3.1 Workflow complet d'une Pull Request

Lifecycle d'une Pull Request

```
WORKFLOW PULL REQUEST COMPLET
══════════════════════════════════════════════════════════════════════

1. FORK (si contribution externe) ou branche (si membre de l'équipe)
   │
   ▼
2. DÉVELOPPEMENT
   git switch -c feature/ma-fonctionnalité
   # ... code ...
   git commit -m "feat: implémentation"
   git push -u origin feature/ma-fonctionnalité
   │
   ▼
3. OUVERTURE DE LA PR (via GitHub ou gh CLI)
   • Titre descriptif
   • Description : contexte, changements, screenshots
   • Labels : feature, bug, documentation...
   • Reviewers : assigner des relecteurs
   • Milestone : version cible
   • Draft PR si travail en cours
   │
   ▼
4. REVIEW (par les relecteurs)
   • Commentaires sur les lignes de code
   • Suggestions de modifications
   • Approbation (Approve) ou demande de changements (Request changes)
   │
   ▼
5. CI/CD (Tests automatiques via GitHub Actions)
   • Tests unitaires
   • Linting, formatage
   • Build
   │
```

```
    ▼
6. MERGE (si tout est approuvé)
   • Merge commit     : préserve les commits
   • Squash and merge : condense en 1 commit
   • Rebase and merge : historique linéaire
    │
    ▼
7. NETTOYAGE
   • Suppression de la branche de feature
   • Fermeture automatique des issues liées
══════════════════════════════════════════════════════════════════
```

```
# Créer une PR via GitHub CLI
$ gh pr create --title "feat: page de connexion" \
               --body "Implémente l'authentification. Closes #42" \
               --base main \
               --head feature/login

# Créer une Draft PR
$ gh pr create --draft

# Voir les PRs
$ gh pr list
$ gh pr view 123
$ gh pr checkout 123   # checkout la branche de la PR localement

# Approuver une PR
$ gh pr review 123 --approve

# Merger une PR
$ gh pr merge 123 --squash
$ gh pr merge 123 --merge
$ gh pr merge 123 --rebase
```

4.3.2 Les trois stratégies de merge de PR

Stratégie	Résultat	Quand l'utiliser
Merge commit	Commit de merge créé, tous les commits préservés	Quand l'historique de la branche a de la valeur
Squash and merge	Tous les commits condensés en UN seul sur main	Feature développée en plusieurs petits commits WIP
Rebase and merge	Commits rejoués sur main, historique linéaire	Commits propres et atomiques dès le départ

4.3.3 Revue de code — Bonnes pratiques

- Relire le code dans son contexte, pas juste les lignes modifiées
- Commenter de manière constructive : expliquer POURQUOI et COMMENT améliorer
- Distinguer les commentaires bloquants (must) des suggestions (nit/optional)
- Utiliser les suggestions de code GitHub pour proposer des modifications directes

- Vérifier : logique, performance, sécurité, tests, documentation

> 💡 **PRs petites = revues meilleures —** Bonne pratique : définir dans CONTRIBUTING.md ce qui est attendu lors d'une revue. Limitez la taille des PRs (idéalement < 400 lignes changées) pour des revues plus efficaces.

4.4 Issues — Suivi de projet

4.4.1 Créer et gérer les Issues

```
# Via GitHub CLI
$ gh issue create --title "Bug: crash à la déconnexion" \
                  --body "Étapes de reproduction:\n1. Se connecter\n2. Cliquer
déconnexion" \
                  --label "bug,priority:high"

$ gh issue list
$ gh issue view 42
$ gh issue close 42 --comment "Corrigé dans PR #45"
```

4.4.2 Labels, Milestones et templates

Élément	Usage	Exemple
Labels	Catégoriser et filtrer	bug, feature, help wanted, good first issue
Milestones	Regrouper pour une version	v2.0.0, Sprint 14
Assignees	Responsable de l'issue	@jean-dupont
Projects	Vue kanban sur les issues	Backlog → In Progress → Done

4.4.3 Références dans les messages de commit

```
# Fermer automatiquement une issue lors du merge de la PR
$ git commit -m "fix: corriger crash déconnexion

Closes #42
Fixes #43
Resolves #44"

# Simplement mentionner (sans fermer)
# "Voir #42" ou "#42" dans le message

# Référencer une issue d'un autre repo
# "Fixes owner/repo#42"
```

4.5 GitHub Pages

Voir la Partie 5 pour une documentation approfondie de GitHub Pages.

4.6 Releases — Publier des versions

4.6.1 Créer une Release

```
# Créer un tag (prérequis)
$ git tag -a v1.0.0 -m "Version 1.0.0"
$ git push origin v1.0.0

# Créer une release via GitHub CLI
$ gh release create v1.0.0 \
  --title "Version 1.0.0 — Première stable" \
  --notes "## Nouveautés\n- Authentification JWT\n- Page de profil" \
  --target main

# Attacher des assets (binaires, archives)
$ gh release upload v1.0.0 ./dist/app-linux-amd64 ./dist/app-windows.exe

# Pré-release
$ gh release create v1.1.0-beta.1 --prerelease

# Lister les releases
$ gh release list
```

4.6.2 Générer un changelog automatique

GitHub peut générer automatiquement un changelog basé sur les PRs mergées. Il suffit de configurer .github/release.yml :

```
# .github/release.yml
changelog:
  exclude:
    labels:
      - ignore-for-release
  categories:
    - title: 🚀 Nouvelles fonctionnalités
      labels:
        - feature
        - enhancement
    - title: 🐛 Corrections de bugs
      labels:
        - bug
    - title: 📚 Documentation
      labels:
        - documentation
```

4.7 GitHub Projects — Gestion de projet

GitHub Projects est un outil de planification Kanban intégré à GitHub. Il permet d'organiser les issues et PRs en tableaux personnalisables.

- Vues : Table, Board (Kanban), Roadmap (Gantt)
- Champs personnalisés : priorité, taille, itération, dates

- Automatisation : déplacer les cartes automatiquement
- Filtres et regroupement avancés

4.8 Organisations GitHub

4.8.1 Structure d'une organisation

Structure organisation

```
Organisation GitHub
═══════════════════════════════════════════════════════
│
├── Teams (groupes d'utilisateurs)
│   ├── @org/backend    → accès repos backend
│   ├── @org/frontend   → accès repos frontend
│   └── @org/admins     → accès complet
│
├── Repositories
│   ├── api-service     (accès: backend)
│   ├── frontend-app    (accès: frontend)
│   └── infra           (accès: admins)
│
└── Paramètres
    ├── Politique SSO (SAML)
    ├── IP allowlist
    ├── Audit log
    └── Secrets d'organisation
═══════════════════════════════════════════════════════
```

Rôle	Permissions
Owner	Accès complet, gestion organisation
Member	Accès aux repos de l'organisation selon les teams
Billing manager	Gestion de la facturation uniquement
Outside collaborator	Accès à des repos spécifiques uniquement

4.8.2 Permissions sur les repositories

Permission	Read	Triage	Write	Maintain	Admin
Cloner/Lire	✓	✓	✓	✓	✓
Issues/PRs	Lire	Gérer	Créer	Gérer	Gérer
Push code	✗	✗	✓	✓	✓
Merger PRs	✗	✗	✓	✓	✓
Gérer branches	✗	✗	✗	✓	✓
Settings repo	✗	✗	✗	✗	✓

Partie 5 : GitHub Pages

5.1 Fonctionnement de GitHub Pages

GitHub Pages est un service d'hébergement de sites statiques directement depuis un dépôt GitHub. Il sert les fichiers HTML, CSS et JavaScript stockés dans une branche ou un répertoire spécifique, via un CDN mondial.

Architecture GitHub Pages

```
ARCHITECTURE GITHUB PAGES
══════════════════════════════════════════════════════════════════════

Votre dépôt GitHub
├── branche: gh-pages  ← source des fichiers statiques
│    ├── index.html
│    ├── style.css
│    └── assets/
│
↓ (GitHub build + CDN)

https://username.github.io/repo-name/

──────────────────────────────────────────────────────────────────────

Flux de déploiement :

Commit → Push → GitHub Pages build → CDN → Accessible publiquement
            ↑
       (Optionnel: Jekyll, Hugo, etc. pour les générateurs de sites)
══════════════════════════════════════════════════════════════════════
```

5.2 Types d'URLs GitHub Pages

Type de site	URL par défaut	Source
Site utilisateur	https://username.github.io	Dépôt: username.github.io
Site organisation	https://orgname.github.io	Dépôt: orgname.github.io
Site de projet	https://username.github.io/projet	N'importe quel dépôt

5.3 Configuration de GitHub Pages

5.3.1 Via l'interface GitHub

1. Aller dans Settings → Pages
2. Choisir la source : branche (gh-pages, main, /docs) ou GitHub Actions

3. Sauvegarder — le site sera disponible en quelques minutes

5.3.2 Via une branche gh-pages

```
# Méthode 1 : Créer manuellement la branche gh-pages
$ git switch --orphan gh-pages   # branche sans historique
$ echo '<h1>Mon site</h1>' > index.html
$ git add index.html
$ git commit -m "Initial GitHub Pages"
$ git push origin gh-pages

# Méthode 2 : Utiliser ghp-import (pour les générateurs de sites)
$ pip install ghp-import
$ mkdocs build
$ ghp-import site -n -p   # -n = no-Jekyll, -p = push
```

5.4 Domaines personnalisés

5.4.1 Configuration DNS

Pour utiliser votre propre domaine (ex: monprojet.com) avec GitHub Pages, vous devez configurer les DNS chez votre registrar.

Configuration DNS

```
Configuration DNS pour domaine apex (monprojet.com) :
══════════════════════════════════════════════════════════════════════
Chez votre registrar, ajouter ces enregistrements A :

@ (apex)   A    185.199.108.153
@ (apex)   A    185.199.109.153
@ (apex)   A    185.199.110.153
@ (apex)   A    185.199.111.153

─── OU pour un sous-domaine (www.monprojet.com) ───

www   CNAME   username.github.io.

─── Puis dans le dépôt GitHub ───

Settings → Pages → Custom domain → "monprojet.com" → Save

GitHub crée automatiquement un fichier CNAME à la racine du site.
══════════════════════════════════════════════════════════════════════
```

5.4.2 HTTPS et certificats SSL

GitHub Pages fournit automatiquement des certificats SSL gratuits via Let's Encrypt pour tous les domaines personnalisés. La case « Enforce HTTPS » dans les paramètres force la redirection HTTP → HTTPS.

> **Délais de propagation —** Après avoir configuré un domaine personnalisé, attendez 24-48h pour la propagation DNS. Le certificat SSL peut mettre jusqu'à 1h à être émis. Vérifiez avec : dig +noall +answer monprojet.com

5.4.3 Le fichier CNAME

```
# Le fichier CNAME (à la racine du site) doit contenir votre domaine
$ cat CNAME
monprojet.com

# GitHub le crée automatiquement depuis les Settings
# Mais vous pouvez aussi le créer manuellement :
$ echo "monprojet.com" > CNAME
$ git add CNAME && git commit -m "docs: configurer domaine personnalisé"
$ git push
```

5.5 Jekyll — Le générateur de sites intégré

GitHub Pages supporte nativement Jekyll, un générateur de sites statiques en Ruby. Il convertit les fichiers Markdown en HTML automatiquement.

```
# Structure d'un site Jekyll
mon-site/
├── _config.yml          ← Configuration Jekyll
├── _layouts/            ← Templates HTML
│   ├── default.html
│   └── post.html
├── _posts/              ← Articles de blog
│   └── 2024-01-15-mon-article.md
├── _includes/           ← Composants réutilisables
├── assets/              ← CSS, JS, images
└── index.md             ← Page d'accueil

# _config.yml minimal :
title: Mon Blog
description: Mes articles
theme: minima          # thème GitHub Pages
baseurl: "/mon-repo"   # vide pour site utilisateur
url: "https://username.github.io"

# Désactiver Jekyll (si site HTML pur)
$ touch .nojekyll
$ git add .nojekyll && git commit -m "disable Jekyll"
```

5.6 Publier avec GitHub Actions (moderne)

La méthode moderne pour publier sur GitHub Pages utilise GitHub Actions, ce qui permet de builder n'importe quel générateur de sites statiques.

```
# .github/workflows/pages.yml — Exemple avec Hugo
name: Deploy Hugo site to Pages

on:
  push:
    branches: ["main"]
  workflow_dispatch:
```

```
permissions:
  contents: read
  pages: write
  id-token: write

concurrency:
  group: "pages"
  cancel-in-progress: false

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          submodules: recursive
          fetch-depth: 0

      - name: Setup Hugo
        uses: peaceiris/actions-hugo@v3
        with:
          hugo-version: 'latest'
          extended: true

      - name: Build with Hugo
        run: hugo --minify

      - name: Upload artifact
        uses: actions/upload-pages-artifact@v3
        with:
          path: ./public

  deploy:
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    runs-on: ubuntu-latest
    needs: build
    steps:
      - name: Deploy to GitHub Pages
        id: deployment
        uses: actions/deploy-pages@v4
```

5.7 Cas pratiques GitHub Pages

5.7.1 Site personnel simple

```
# 1. Créer le repo username.github.io
# 2. Cloner localement
$ git clone git@github.com:username/username.github.io.git
$ cd username.github.io

# 3. Créer index.html
$ cat > index.html << 'EOF'
```

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Mon Portfolio</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <h1>Bonjour, je suis [Votre Nom]</h1>
  <p>Développeur passionné...</p>
</body>
</html>
EOF

# 4. Pusher
$ git add . && git commit -m "Initial site" && git push
# Site disponible sur https://username.github.io en quelques minutes
```

5.7.2 Documentation avec MkDocs

```
# 1. Installer MkDocs
$ pip install mkdocs mkdocs-material

# 2. Initialiser
$ mkdocs new mon-projet-docs
$ cd mon-projet-docs

# 3. Configurer mkdocs.yml
site_name: Mon Projet
theme:
  name: material
  language: fr

nav:
  - Accueil: index.md
  - Guide: guide.md
  - API: api.md

# 4. Workflow GitHub Actions
# .github/workflows/docs.yml
name: Deploy docs
on:
  push:
    branches: ["main"]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: 3.x
      - run: pip install mkdocs-material
      - run: mkdocs gh-deploy --force
```

5.7.3 Erreurs fréquentes GitHub Pages

Erreur	Cause	Solution
404 sur toutes les pages	baseurl non configuré pour les sous-repos	Ajouter baseurl: /nom-repo dans _config.yml
CSS non chargé	Chemins relatifs incorrects	Utiliser {{site.baseurl}}/assets/style.css
Certificat SSL invalide	Propagation DNS incomplète	Attendre 24-48h, vérifier avec dig
Build jekyll échoue	Gems incompatibles	Ajouter .nojekyll ou corriger Gemfile
Domaine custom revient à défaut	CNAME écrasé par GitHub	Versionner le fichier CNAME dans le repo

Partie 6 : GitHub Actions — CI/CD

6.1 Concepts fondamentaux

6.1.1 CI/CD — Définitions

Concept	Définition	Exemples
CI — Intégration Continue	Automatiser les tests et vérifications à chaque push	Tests unitaires, linting, analyse statique
CD — Déploiement Continu	Automatiser le déploiement après validation	Deploy en staging, prod, GitHub Pages
Runner	Machine virtuelle qui exécute les workflows	ubuntu-latest, windows-latest, macos-latest
Workflow	Fichier YAML définissant l'automatisation	.github/workflows/ci.yml
Job	Ensemble d'étapes s'exécutant sur un runner	test, build, deploy
Step	Commande ou action dans un job	run: npm test, uses: actions/checkout@v4
Action	Brique réutilisable (code Node.js ou Docker)	actions/checkout, actions/setup-node

6.1.2 Architecture complète de GitHub Actions

Architecture GitHub Actions

```
ARCHITECTURE GITHUB ACTIONS
════════════════════════════════════════════════════════════════════════

Événement déclencheur
(push, pull_request, schedule, workflow_dispatch...)
        │
        ▼
┌──────────────────────────────────────────────────────────────────┐
│  WORKFLOW (.github/workflows/ci.yml)                             │
│                                                                  │
│  ┌───────────────────────┐   ┌────────────────────────────────┐  │
│  │  Job: test            │   │  Job: deploy                   │  │
│  │  runs-on: ubuntu      │   │  needs: [test]  ← dépendance   │  │
│  │                       │   │  runs-on: ubuntu               │  │
│  │  Step 1: checkout     │   │                                │  │
│  │  Step 2: setup node   │   │  Step 1: checkout              │  │
│  │  Step 3: install      │   │  Step 2: build                 │  │
```

```
│  │  Step 4: test          │    │  Step 3: deploy                   │ │
│  └────────────────────────┘    └───────────────────────────────────┘ │
│                                                                      │
│  Jobs parallèles par défaut, séquentiels avec 'needs'               │
└──────────────────────────────────────────────────────────────────────┘
       │                              │
       ▼                              ▼
Runner GitHub hébergé         Runner self-hosted
(GitHub paie l'infra)         (votre propre serveur)
══════════════════════════════════════════════════════════════════════════
```

6.2 Syntaxe YAML complète

6.2.1 Déclencheurs (on:)

```
on:
  # Déclencher sur push
  push:
    branches: ["main", "develop"]
    paths: ["src/**", "tests/**"]      # seulement si ces chemins changent
    paths-ignore: ["docs/**", "*.md"]  # ignorer ces chemins

  # Déclencher sur PR
  pull_request:
    branches: ["main"]
    types: [opened, synchronize, reopened]

  # Planification (cron)
  schedule:
    - cron: '0 6 * * 1'  # Chaque lundi à 6h UTC

  # Déclenchement manuel
  workflow_dispatch:
    inputs:
      environment:
        description: "Environnement cible"
        required: true
        default: "staging"
        type: choice
        options: ["staging", "production"]

  # Déclenché par un autre workflow
  workflow_call:
    inputs:
      version:
        required: true
        type: string

  # Événements repository
  release:
    types: [published]

  issues:
    types: [opened, labeled]
```

6.2.2 Contextes et expressions

```
# Contextes disponibles dans les expressions
${{ github.sha }}           # SHA du commit
${{ github.ref }}           # "refs/heads/main"
${{ github.event_name }}    # "push", "pull_request"...
${{ github.actor }}         # login de l'utilisateur
${{ github.repository }}    # "owner/repo"
${{ github.run_number }}    # numéro d'exécution
${{ github.run_id }}        # ID unique de l'exécution

${{ secrets.MY_SECRET }}    # Secret du repository
${{ vars.MY_VAR }}          # Variable du repository (non secrète)
${{ env.MY_ENV }}           # Variable d'environnement définie dans le workflow

${{ runner.os }}            # "Linux", "Windows", "macOS"
${{ matrix.python-version }}  # valeur de la matrice courante

# Expressions conditionnelles
if: github.ref == 'refs/heads/main'
if: github.event_name == 'push'
if: success()               # étapes précédentes OK
if: failure()               # une étape a échoué
if: always()                # s'exécute même en cas d'échec
if: cancelled()             # workflow annulé
if: startsWith(github.ref, 'refs/tags/')  # déclenchement par tag
```

6.2.3 Variables d'environnement et Secrets

```
env:
  # Variables globales (tout le workflow)
  NODE_ENV: production
  API_URL: https://api.exemple.com

jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      # Variables au niveau job
      DEPLOY_ENV: staging
    steps:
      - name: Ma step
        env:
          # Variables au niveau step (priorité la plus haute)
          SECRET_KEY: ${{ secrets.SECRET_KEY }}
          DATABASE_URL: ${{ secrets.DATABASE_URL }}
        run: |
          echo "Deploy to $DEPLOY_ENV"
          ./deploy.sh
```

6.3 Cas pratiques — Workflows complets

6.3.1 Workflow Python complet

```
# .github/workflows/python-ci.yml
name: Python CI

on:
  push:
    branches: ["main", "develop"]
  pull_request:
    branches: ["main"]

jobs:
  lint:
    name: Linting & Formatage
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.12'

      - name: Install linting tools
        run: pip install flake8 black isort mypy

      - name: Check formatting (black)
        run: black --check .

      - name: Check imports (isort)
        run: isort --check .

      - name: Lint (flake8)
        run: flake8 . --max-line-length=100

      - name: Type checking (mypy)
        run: mypy src/

  test:
    name: Tests (${{ matrix.python-version }})
    runs-on: ubuntu-latest
    needs: lint
    strategy:
      matrix:
        python-version: ["3.10", "3.11", "3.12"]
      fail-fast: false    # continuer même si une version échoue

    steps:
      - uses: actions/checkout@v4

      - name: Setup Python ${{ matrix.python-version }}
        uses: actions/setup-python@v5
```

```yaml
        with:
          python-version: ${{ matrix.python-version }}
          cache: 'pip'    # cache des dépendances pip

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install -r requirements-dev.txt

      - name: Run tests with coverage
        run: pytest --cov=src --cov-report=xml --cov-report=term

      - name: Upload coverage
        uses: codecov/codecov-action@v4
        with:
          token: ${{ secrets.CODECOV_TOKEN }}
          file: ./coverage.xml

  security:
    name: Audit de sécurité
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - run: pip install safety bandit
      - run: safety check -r requirements.txt
      - run: bandit -r src/ -ll
```

6.3.2 Workflow Node.js complet

```yaml
# .github/workflows/node-ci.yml
name: Node.js CI/CD

on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [18.x, 20.x, 22.x]

    steps:
      - uses: actions/checkout@v4

      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}
```

```yaml
          cache: 'npm'

      - name: Install dependencies
        run: npm ci           # préférer ci à install en CI

      - name: Lint
        run: npm run lint

      - name: Build
        run: npm run build

      - name: Test
        run: npm test -- --coverage

      - name: Upload build artifacts
        uses: actions/upload-artifact@v4
        with:
          name: build-${{ matrix.node-version }}
          path: dist/
          retention-days: 7

  deploy:
    needs: build-and-test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'   # uniquement sur main
    environment: production               # environnement protégé

    steps:
      - uses: actions/checkout@v4

      - name: Download artifact
        uses: actions/download-artifact@v4
        with:
          name: build-20.x
          path: dist/

      - name: Deploy to server
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.SERVER_HOST }}
          username: ${{ secrets.SERVER_USER }}
          key: ${{ secrets.SSH_PRIVATE_KEY }}
          script: |
            cd /var/www/app
            rsync -av dist/ ./dist/
            pm2 restart app
```

6.3.3 Workflow Java/Maven

```yaml
# .github/workflows/java-ci.yml
name: Java CI with Maven

on:
  push:
    branches: ["main"]
```

```
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up JDK 21
        uses: actions/setup-java@v4
        with:
          java-version: '21'
          distribution: 'temurin'
          cache: maven

      - name: Build with Maven
        run: mvn -B package --no-transfer-progress

      - name: Run tests
        run: mvn -B test

      - name: Generate coverage report
        run: mvn jacoco:report

      - name: Check code quality
        run: mvn sonar:sonar
        env:
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}

      - name: Build Docker image
        if: github.ref == 'refs/heads/main'
        run: |
          docker build -t my-app:${{ github.sha }} .
          docker tag my-app:${{ github.sha }} ghcr.io/${{ github.repository
}}:latest

      - name: Push to GitHub Container Registry
        if: github.ref == 'refs/heads/main'
        run: |
          echo ${{ secrets.GITHUB_TOKEN }} | docker login ghcr.io -u ${{
github.actor }} --password-stdin
          docker push ghcr.io/${{ github.repository }}:latest
```

6.3.4 Workflow Flutter

```
# .github/workflows/flutter-ci.yml
name: Flutter CI

on:
  push:
    branches: ["main"]
  pull_request:

jobs:
```

```yaml
  analyze-and-test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Setup Flutter
        uses: subosito/flutter-action@v2
        with:
          flutter-version: '3.22.0'
          channel: 'stable'
          cache: true

      - name: Install dependencies
        run: flutter pub get

      - name: Verify formatting
        run: dart format --output=none --set-exit-if-changed .

      - name: Analyze project source
        run: flutter analyze

      - name: Run tests
        run: flutter test --coverage

      - name: Upload coverage
        uses: codecov/codecov-action@v4
        with:
          token: ${{ secrets.CODECOV_TOKEN }}

  build-android:
    needs: analyze-and-test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
      - uses: actions/checkout@v4
      - uses: subosito/flutter-action@v2
        with:
          flutter-version: '3.22.0'

      - name: Setup signing
        run: |
          echo "${{ secrets.KEYSTORE_BASE64 }}" | base64 --decode >
android/app/release.keystore

      - name: Build APK
        run: flutter build apk --release
        env:
          KEY_STORE_PASSWORD: ${{ secrets.KEY_STORE_PASSWORD }}
          KEY_PASSWORD: ${{ secrets.KEY_PASSWORD }}
          ALIAS: ${{ secrets.ALIAS }}

      - name: Upload APK
        uses: actions/upload-artifact@v4
```

```
        with:
          name: release-apk
          path: build/app/outputs/flutter-apk/app-release.apk
```

6.3.5 Fonctionnalités avancées de GitHub Actions

```
# Cache de dépendances (accélérer les builds)
- name: Cache pip packages
  uses: actions/cache@v4
  with:
    path: ~/.cache/pip
    key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements.txt') }}
    restore-keys: |
      ${{ runner.os }}-pip-

# Artifacts — Partager des fichiers entre jobs
- name: Upload build output
  uses: actions/upload-artifact@v4
  with:
    name: my-artifact
    path: dist/
    retention-days: 5

- name: Download artifact
  uses: actions/download-artifact@v4
  with:
    name: my-artifact
    path: dist/

# Concurrence — Annuler les anciennes exécutions
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

# Permissions minimales (sécurité)
permissions:
  contents: read
  checks: write
  pull-requests: write

# Outputs entre steps et jobs
jobs:
  setup:
    outputs:
      version: ${{ steps.get-version.outputs.version }}
    steps:
      - id: get-version
        run: echo "version=$(cat VERSION)" >> $GITHUB_OUTPUT

  deploy:
    needs: setup
    steps:
      - run: echo "Deploying version ${{ needs.setup.outputs.version }}"
```

6.3.6 Sécurité dans GitHub Actions

🔴 **Épingler les actions tierces aux hashes —** Toujours épingler les actions tierces à un hash de commit complet (uses: actions/checkout@a5ac7e51b41094c92402da3b24376905380afc29) plutôt qu'à un tag (uses: actions/checkout@v4). Un tag peut être réassigné par l'éditeur pour pointer vers un code malveillant.

```
# ❌ Risqué : le tag peut pointer vers un nouveau code
uses: some-action@v2

# ☑ Sécurisé : le hash est immuable
uses: some-action@a5ac7e51b41094c92402da3b24376905380afc29

# Éviter les injections de script
# ❌ Dangereux si le titre de PR contient du code malveillant
- run: echo "${{ github.event.pull_request.title }}"

# ☑ Sûr : passer via variable d'environnement
- name: Safe echo
  env:
    PR_TITLE: ${{ github.event.pull_request.title }}
  run: echo "$PR_TITLE"
```

Partie 7 : Sécurité

7.1 Authentification : SSH vs HTTPS

Critère	SSH	HTTPS
Authentification	Paire de clés publique/privée	Token PAT ou mot de passe
Sécurité	Très élevée (cryptographie asymétrique)	Élevée (TLS + token)
Configuration	Une fois par machine	Par machine ou gestionnaire
Recommandé pour	Développement quotidien	Scripts, CI/CD, outils
Authentification MFA	La clé privée remplace le MFA	Token PAT requis si MFA activé
URL format	git@github.com:user/repo.git	https://github.com/user/repo.git

7.2 Configuration des clés SSH

7.2.1 Générer une paire de clés SSH

```
# Générer une clé ED25519 (recommandée — plus récente et sécurisée)
$ ssh-keygen -t ed25519 -C "votre@email.com"

# Si ED25519 non supporté (systèmes anciens) :
$ ssh-keygen -t rsa -b 4096 -C "votre@email.com"

# Lors de la génération :
# Emplacement : ~/.ssh/id_ed25519 (appuyer Entrée pour défaut)
# Passphrase : fortement recommandée (chiffre la clé privée)

# Résultat :
# ~/.ssh/id_ed25519      ← clé PRIVÉE (ne JAMAIS partager)
# ~/.ssh/id_ed25519.pub  ← clé PUBLIQUE (à donner à GitHub)
```

> 💡 **Toujours utiliser une passphrase —** La passphrase chiffre votre clé privée. Même si quelqu'un vole votre fichier de clé, il ne peut pas l'utiliser sans la passphrase. Utilisez ssh-agent pour ne pas la retaper à chaque fois.

7.2.2 Configurer ssh-agent

```
# Démarrer ssh-agent
$ eval "$(ssh-agent -s)"

# Ajouter votre clé à l'agent (mémorise la passphrase)
$ ssh-add ~/.ssh/id_ed25519
```

```
# Sur macOS, ajouter au trousseau automatiquement
$ ssh-add --apple-use-keychain ~/.ssh/id_ed25519

# Sur Linux, configurer ~/.bashrc ou ~/.zshrc pour démarrage auto
if [ -z "$SSH_AUTH_SOCK" ]; then
   eval "$(ssh-agent -s)"
   ssh-add ~/.ssh/id_ed25519
fi
```

7.2.3 Ajouter la clé publique à GitHub

```
# Afficher la clé publique
$ cat ~/.ssh/id_ed25519.pub
# Copier la sortie (commence par "ssh-ed25519 AAAA...")

# Via GitHub CLI
$ gh ssh-key add ~/.ssh/id_ed25519.pub --title "MacBook Pro Travail"

# Via l'interface :
# GitHub → Settings → SSH and GPG keys → New SSH key
# Title: descriptif (ex: "MacBook Pro Travail 2024")
# Key type: Authentication Key
# Coller la clé publique

# Tester la connexion
$ ssh -T git@github.com
# Hi username! You've successfully authenticated, but GitHub does
# not provide shell access.
```

7.2.4 Gérer plusieurs comptes GitHub

```
# ~/.ssh/config — Gérer plusieurs identités SSH
Host github-perso
    HostName github.com
    User git
    IdentityFile ~/.ssh/id_ed25519_perso
    IdentitiesOnly yes

Host github-travail
    HostName github.com
    User git
    IdentityFile ~/.ssh/id_ed25519_travail
    IdentitiesOnly yes

# Utilisation :
# Perso : git@github-perso:mon-user/repo.git
# Travail : git@github-travail:entreprise/repo.git

# Changer l'URL d'un repo existant
$ git remote set-url origin git@github-travail:entreprise/repo.git
```

7.3 Personal Access Tokens (PAT)

7.3.1 Créer un PAT

Les PAT remplacent les mots de passe GitHub pour l'authentification HTTPS et les API calls. Il existe deux types : Classic et Fine-grained.

```
# GitHub → Settings → Developer settings → Personal access tokens

# PAT Classic :
# • Accès à tous les repos auxquels vous avez accès
# • Simple mais moins granulaire
# • Portée (scope) : repo, workflow, packages...

# PAT Fine-grained (recommandé) :
# • Scope limité à des repos spécifiques
# • Permissions précises (read, write par ressource)
# • Date d'expiration obligatoire
# • Plus de sécurité

# Utiliser un PAT pour les opérations HTTPS
$ git clone https://github.com/user/repo.git
Username: votre_username
Password: ghp_votre_token_ici   # PAT, pas le mot de passe

# Stocker via credential helper
$ git config --global credential.helper store    # stockage en clair (Linux)
$ git config --global credential.helper osxkeychain  # macOS
$ git config --global credential.helper manager  # Windows
```

7.4 Secrets GitHub

7.4.1 Repository Secrets

```
# Ajouter un secret via GitHub CLI
$ gh secret set MY_SECRET --body "valeur_secrète"
$ gh secret set DATABASE_URL < .env.production   # depuis un fichier

# Lister les secrets
$ gh secret list

# Dans un workflow
steps:
  - name: Deploy
    env:
      DB_URL: ${{ secrets.DATABASE_URL }}
      API_KEY: ${{ secrets.API_KEY }}
    run: ./deploy.sh
```

7.4.2 Organisation et Environment Secrets

```
# Organisation secrets : partagés entre tous les repos de l'org
# Repository secrets : propres à un repo
# Environment secrets : propres à un environnement de déploiement

# Exemple d'environnement protégé
jobs:
  deploy-production:
    environment:
      name: production            # Environnement protégé
      url: https://mon-app.com
    runs-on: ubuntu-latest
    steps:
      - name: Deploy
        env:
          PROD_KEY: ${{ secrets.PROD_DEPLOY_KEY }}  # secret d'env
        run: ./deploy-prod.sh

# Les environnements peuvent avoir :
# • Des reviewers requis (approbation manuelle)
# • Des règles de déploiement (branches autorisées)
# • Des secrets spécifiques
```

7.4.3 Signer les commits GPG

```
# Générer une clé GPG
$ gpg --full-generate-key
# Type: RSA 4096 bits, durée: 2 ans

# Obtenir l'ID de la clé
$ gpg --list-secret-keys --keyid-format=long
/Users/user/.gnupg/secring.gpg
sec   4096R/3AA5C34371567BD2 2024-01-01 [expires: 2026-01-01]

# Configurer Git pour signer avec cette clé
$ git config --global user.signingkey 3AA5C34371567BD2
$ git config --global commit.gpgsign true   # signer tous les commits

# Signer un commit manuellement
$ git commit -S -m "feat: fonctionnalité signée"

# Exporter la clé publique vers GitHub
$ gpg --armor --export 3AA5C34371567BD2
# GitHub → Settings → SSH and GPG keys → New GPG key
```

Partie 8 : Workflows Professionnels

8.1 Git Flow

Git Flow est un modèle de branching formalisé par Vincent Driessen en 2010. Il définit des branches avec des rôles précis, adapté aux projets avec des cycles de release planifiés.

Git Flow

```
GIT FLOW — VUE D'ENSEMBLE
═══════════════════════════════════════════════════════════════════════

main       ──────●──────────────────────────────────●──────────▶ (production)
                 │ tag: v1.0                         │ tag: v1.1
                 │                                   │
hotfix     ──────│──────────●──────────────────────▶│──────────
                 │           │ hotfix/bug-urgnt       │
                 │           │                        │
release    ──────│───────────│───────●───────────────│──────────
                 │           │        │ release/v1.1  │
                 │           │        │               │
develop    ──●──────────────────────────────────────────────▶ (intégration)
              │   │          │        │ ← merge        │
              │   └──────────┘        │                │
feature    ───────────────────────────●───────────────────▶
             feature/login           feature/payment

───────────────────────────────────────────────────────────────────────
Branches permanentes :
• main/master  : code en production, protégé, toujours stable
• develop      : intégration des nouvelles fonctionnalités

Branches temporaires :
• feature/*    : nouvelles fonctionnalités (depuis develop → develop)
• release/*    : préparation d'une release (depuis develop → main+develop)
• hotfix/*     : corrections urgentes en prod (depuis main → main+develop)
═══════════════════════════════════════════════════════════════════════
```

```
# Installation de git-flow (outil CLI)
$ brew install git-flow-avh    # macOS
$ apt-get install git-flow     # Linux

# Initialiser Git Flow dans un repo
$ git flow init
# Branch names:
# Production: main
# Development: develop
# Feature prefix: feature/
# Release prefix: release/
# Hotfix prefix: hotfix/

# ── Feature ──────────────────────────────
```

```
# Démarrer une feature
$ git flow feature start nom-fonctionnalite
# Équivalent : git checkout -b feature/nom-fonctionnalite develop

# Finir une feature (merge dans develop)
$ git flow feature finish nom-fonctionnalite

# ── Release ──────────────────────────────
# Démarrer une release
$ git flow release start 1.1.0
# Équivalent : git checkout -b release/1.1.0 develop

# Bumper la version, corriger les bugs mineurs, mettre à jour le CHANGELOG...
$ git flow release finish 1.1.0
# Merge dans main ET develop, crée le tag v1.1.0

# ── Hotfix ───────────────────────────────
$ git flow hotfix start bug-critique
$ git flow hotfix finish bug-critique
# Merge dans main ET develop
```

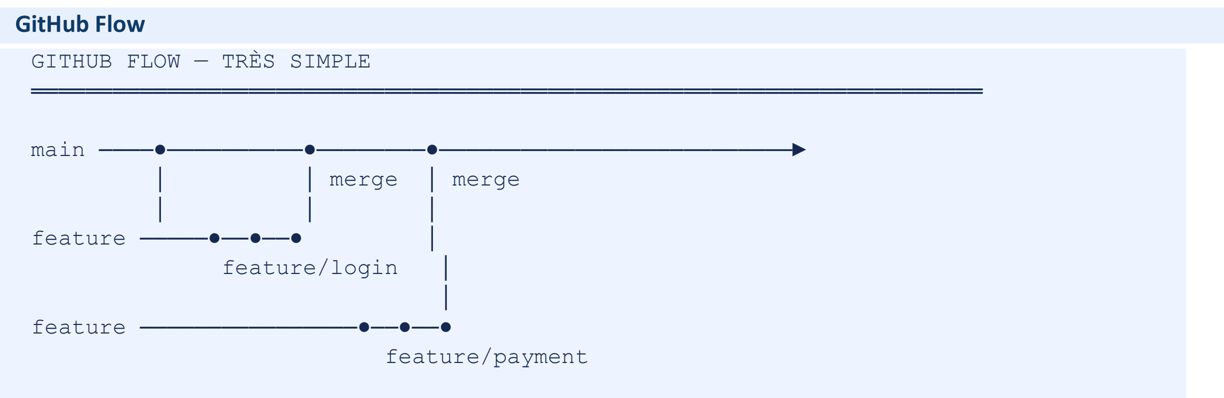
8.1.1 Avantages et inconvénients de Git Flow

Avantages	Inconvénients
Structure claire et documentée	Complexité élevée pour les petites équipes
Adapté aux releases versionnées	Historique non-linéaire difficile à lire
Isolation parfaite des features	Long temps de vie des branches → conflits
Support des hotfixes en production	Pas adapté au déploiement continu
Processus bien connu des équipes	Merge hell possible sur de gros projets

8.2 GitHub Flow

GitHub Flow est un workflow simplifié recommandé par GitHub pour les projets avec déploiement continu. Il n'a qu'une seule branche principale (main) et des branches courtes de fonctionnalités.

GitHub Flow



```
Règles :
1. main est TOUJOURS déployable en production
2. Créer une branche pour chaque tâche
3. Commiter fréquemment
4. Ouvrir une PR quand prêt pour review
5. Merger dans main via PR après approbation
6. Déployer main immédiatement après merge
══════════════════════════════════════════════════════════════════════
```

```
# GitHub Flow en pratique
# 1. Synchroniser avec main
$ git switch main && git pull

# 2. Créer une branche de feature
$ git switch -c feature/nouveau-dashboard

# 3. Développer et commiter régulièrement
$ git add . && git commit -m "feat: ajouter widget statistiques"

# 4. Pousser et ouvrir une PR
$ git push -u origin feature/nouveau-dashboard
$ gh pr create --title "feat: nouveau dashboard" --base main

# 5. Après review et approbation : merger
$ gh pr merge --squash

# 6. Nettoyer
$ git switch main && git pull
$ git branch -d feature/nouveau-dashboard
```

8.3 Trunk Based Development

Trunk Based Development (TBD) est le modèle le plus radical : tous les développeurs commitent directement sur la branche principale (trunk/main) plusieurs fois par jour. C'est le modèle utilisé par Google et d'autres grandes entreprises tech.

Trunk Based Development

```
TRUNK BASED DEVELOPMENT
══════════════════════════════════════════════════════════════════════

trunk/main ──●──●──●──●──●──●──●──●──●──●──●──▶
                  ↑  ↑     ↑     ↑     ↑
             Dev A B  B    A     C   A  B
             commits toutes les quelques heures !

Feature flags pour les fonctionnalités en cours :

if (featureFlags.isEnabled('new-dashboard', user)):
    render(NewDashboard)
else:
    render(OldDashboard)

Branches de release UNIQUEMENT pour les corrections :
release/v1.1 ──────●─────────────────────────────
```

```
                    cherry-pick de corrections seulement
═══════════════════════════════════════════════════════════════════
```

8.3.1 Prérequis pour TBD

- Suite de tests complète et rapide (< 10 min)
- CI/CD mature avec déploiement automatique
- Feature flags pour masquer les fonctionnalités incomplètes
- Équipe avec discipline et habitude des petits commits
- Pair programming ou revues de code asynchrones rapides

8.4 Comparaison des trois workflows

Critère	Git Flow	GitHub Flow	Trunk Based Dev
Complexité	Élevée	Faible	Faible + discipline
Branches permanentes	main + develop	main	main uniquement
Durée des branches	Semaines	Jours	Heures
Fréquence de deploy	Planifiée	Après chaque PR	Continue (pluri-quotidienne)
Idéal pour	Software packagé, releases planifiées	Apps web, SaaS, petites équipes	Grande équipe, maturité CI/CD élevée
Feature flags	Non requis	Optionnel	Requis
Tests requis	Recommandés	Fortement recommandés	Indispensables

Partie 9 : Cas Pratiques Complets

9.1 Créer un projet personnel de A à Z

````
# 1. Créer le repo sur GitHub
$ gh repo create mon-projet --public --description "Mon super projet"

# 2. Cloner en local
$ git clone git@github.com:username/mon-projet.git
$ cd mon-projet

# 3. Structure initiale
$ mkdir -p src tests docs .github/workflows
$ cat > README.md << 'EOF'
# Mon Projet

Description courte du projet.

## Installation
```bash
pip install -r requirements.txt
```

## Usage
```python
from mon_projet import main
main()
```
EOF

$ cat > .gitignore << 'EOF'
__pycache__/
*.pyc
venv/
.env
dist/
*.egg-info/
EOF

$ cat > requirements.txt << 'EOF'
requests==2.31.0
pytest==7.4.0
EOF

$ cat > src/main.py << 'EOF'
def main():
    print("Hello, World!")

if __name__ == "__main__":
    main()
````

```
EOF

# 4. Premier commit
$ git add .
$ git commit -m "chore: initialisation du projet"
$ git push -u origin main

# 5. Protéger la branche main (via GitHub)
# Settings → Branches → Add rule → main
# ✓ Require a pull request before merging
# ✓ Require approvals: 1
# ✓ Require status checks to pass

# 6. Ajouter le CI
$ cat > .github/workflows/ci.yml << 'EOF'
name: CI
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - run: pip install -r requirements.txt
      - run: pytest
EOF

$ git add .github/
$ git commit -m "ci: ajouter workflow GitHub Actions"
$ git push
```

9.2 Publier un site GitHub Pages complet

```
# 1. Créer le repo username.github.io
$ gh repo create username.github.io --public
$ git clone git@github.com:username/username.github.io.git
$ cd username.github.io

# 2. Structure HTML/CSS/JS
$ mkdir -p assets/css assets/js
$ cat > index.html << 'EOF'
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Portfolio - Username</title>
  <link rel="stylesheet" href="assets/css/style.css">
</head>
<body>
  <header><h1>Mon Portfolio</h1></header>
```

```
  <main>
    <section id="about">
      <h2>À propos</h2>
      <p>Développeur passionné...</p>
    </section>
    <section id="projects">
      <h2>Projets</h2>
      <!-- Liste des projets -->
    </section>
  </main>
  <script src="assets/js/main.js"></script>
</body>
</html>
EOF

# 3. Push et activer GitHub Pages
$ git add . && git commit -m "feat: site portfolio initial"
$ git push -u origin main
# Dans Settings → Pages → Source: main branch

# 4. Domaine personnalisé (optionnel)
$ echo "monsite.com" > CNAME
$ git add CNAME && git commit -m "docs: domaine personnalisé"
$ git push
```

9.3 Contribuer à un projet Open Source

Processus de contribution OS

```
WORKFLOW CONTRIBUTION OPEN SOURCE
══════════════════════════════════════════════════════════════════════

1. EXPLORER le projet
   • Lire README, CONTRIBUTING, CODE_OF_CONDUCT
   • Chercher les issues "good first issue" ou "help wanted"
   • Rejoindre les discussions si nécessaire avant de coder

2. FORK le dépôt (sur GitHub)
   → Crée votre copie : vous/nom-projet

3. CLONER votre fork
   $ git clone git@github.com:vous/nom-projet.git
   $ cd nom-projet

4. CONFIGURER upstream
   $ git remote add upstream git@github.com:original/nom-projet.git

5. CRÉER une branche dédiée
   $ git switch -c fix/description-du-bug

6. IMPLÉMENTER le changement
   • Respecter le style de code existant
   • Ajouter/mettre à jour les tests
   • Mettre à jour la documentation

7. SYNCHRONISER avant PR (éviter les conflits)
```

```
   $ git fetch upstream
   $ git rebase upstream/main

8. POUSSER et créer la PR
   $ git push -u origin fix/description-du-bug
   $ gh pr create --repo original/nom-projet \
     --title "fix: décrire la correction" \
     --body "Closes #42\n\n## Description\n..."

9. RÉPONDRE aux reviews
   • Faire les modifications demandées
   $ git add . && git commit -m "review: appliquer suggestions"
   $ git push

10. MERGE (par le mainteneur du projet)
══════════════════════════════════════════════════════════════════════
```

```
# Commandes clés pour la contribution open source

# Synchroniser son fork avec l'upstream
$ git fetch upstream
$ git switch main
$ git merge upstream/main    # ou : git rebase upstream/main
$ git push origin main

# Garder sa branche de feature à jour
$ git switch feature/ma-feature
$ git rebase upstream/main
$ git push --force-with-lease origin feature/ma-feature

# Vérifier l'état de sa PR
$ gh pr status
$ gh pr view

# Répondre à un commentaire de review
$ git add -p    # stager les modifications précises
$ git commit -m "refactor: extraire fonction selon suggestion review"
$ git push
```

9.4 Travail en équipe — Workflow complet

```
# ─── CHAQUE MATIN ───────────────────────────────────────────────
# 1. Synchroniser main
$ git switch main
$ git pull --ff-only   # fast-forward seulement, sûr

# 2. Synchroniser sa branche feature en cours
$ git switch feature/mon-travail
$ git rebase main

# ─── PENDANT LE DÉVELOPPEMENT ────────────────────────────────────
# 3. Commits atomiques et fréquents
$ git add -p                        # stager précisément
$ git commit -m "feat: ajouter validation email"
```

```
# ─── AVANT CHAQUE PUSH ──────────────────────────────────────────
# 4. Vérifier ce qu'on va pousser
$ git log origin/feature/mon-travail..HEAD --oneline
$ git diff origin/feature/mon-travail

# 5. Pousser
$ git push   # ou git push -u origin feature/mon-travail (1ère fois)

# ─── OUVRIR UNE PULL REQUEST ────────────────────────────────────
# 6. Créer la PR
$ gh pr create --title "feat: validation email" \
               --body "## Description\nAjoute la validation...\n\nCloses #23"

# ─── REVUE DE CODE ──────────────────────────────────────────────
# 7. Reviewer une PR de collègue
$ gh pr checkout 42              # basculer sur la branche de la PR
$ npm test                       # vérifier que les tests passent
$ gh pr review 42 --approve
# ou
$ gh pr review 42 --request-changes --body "Peux-tu extraire cette logique ?"

# ─── APRÈS LE MERGE ─────────────────────────────────────────────
# 8. Nettoyer
$ git switch main && git pull
$ git branch -d feature/mon-travail
$ git remote prune origin    # supprimer les tracking branches obsolètes
```

Partie 10 : Référence Rapide

10.1 Tableau des commandes Git

Commande	Description	Exemple
`git init`	Initialiser un dépôt	`git init mon-projet`
`git clone`	Cloner un dépôt distant	`git clone git@github.com:u/r.git`
`git status`	État des fichiers	`git status -s`
`git add`	Stager des modifications	`git add -p`
`git commit`	Créer un commit	`git commit -m "feat: ..."`
`git log`	Historique des commits	`git log --oneline --graph --all`
`git diff`	Voir les modifications	`git diff --staged`
`git show`	Inspecter un commit/objet	`git show abc123`
`git branch`	Gérer les branches	`git branch -a`
`git switch`	Changer de branche	`git switch -c feature/x`
`git merge`	Fusionner des branches	`git merge --no-ff feature/x`
`git rebase`	Rebaser une branche	`git rebase -i HEAD~3`
`git cherry-pick`	Appliquer un commit spécifique	`git cherry-pick abc123`
`git reset`	Annuler/réécrire l'historique	`git reset --soft HEAD~1`
`git revert`	Créer un commit d'annulation	`git revert HEAD`
`git stash`	Sauvegarder temporairement	`git stash push -m "WIP"`
`git tag`	Gérer les tags	`git tag -a v1.0.0 -m "..."`
`git remote`	Gérer les remotes	`git remote add upstream URL`
`git fetch`	Télécharger sans merger	`git fetch --all --prune`
`git pull`	Télécharger et merger	`git pull --rebase`
`git push`	Envoyer vers le distant	`git push --force-with-lease`
`git reflog`	Journal des déplacements HEAD	`git reflog`
`git blame`	Historique ligne par ligne	`git blame -L 10,20 fichier.py`
`git bisect`	Trouver le commit bugué	`git bisect start/good/bad`
`git worktree`	Travailler sur plusieurs branches simultanément	`git worktree add ../fix main`

`git submodule`	Gérer les sous-modules	`git submodule update --init`
`git clean`	Supprimer les fichiers non trackés	`git clean -fd`
`git restore`	Restaurer des fichiers	`git restore --staged fichier`
`git grep`	Chercher dans le code	`git grep -n "TODO"`

10.2 Erreurs fréquentes et solutions

Erreur : src refspec main does not match any

```
$ git push origin main
error: src refspec main does not match any.

# Cause : La branche "main" n'existe pas (branche nommée "master" ou pas de
commit)
# Solutions :
# 1. Faire un premier commit
$ git add . && git commit -m "Initial commit"
$ git push -u origin main

# 2. Renommer la branche courante
$ git branch -m master main
$ git push -u origin main
```

Erreur : non-fast-forward (rejected)

```
$ git push
! [rejected] main -> main (non-fast-forward)
error: failed to push some refs

# Cause : Quelqu'un a pushé pendant que vous travailliez
# Solutions :
$ git pull --rebase    # récupérer et rebaser (historique linéaire)
$ git push             # repousser

# OU si vous ne voulez pas rebaser
$ git pull             # récupérer et merger
$ git push
```

Erreur : merge conflict

```
CONFLICT (content): Merge conflict in src/app.py
Automatic merge failed; fix conflicts and then commit.

# Cause : Les deux branches ont modifié les mêmes lignes
# Résolution :
$ git status                    # voir les fichiers en conflit
# Éditer les fichiers, supprimer les marqueurs <<< === >>>
$ git add src/app.py            # stager la résolution
$ git commit                    # finaliser le merge
```

```
# OU annuler :
$ git merge --abort
```

Erreur : detached HEAD

```
HEAD detached at abc123

# Cause : Checkout vers un commit plutôt qu'une branche
# Si vous avez committé en detached HEAD :
$ git branch sauvegarde         # créer une branche sur ce commit
$ git switch main               # revenir sur une branche normale

# Si vous n'avez pas committé :
$ git switch main               # simplement revenir
```

Erreur : permission denied (publickey)

```
$ git push
Permission denied (publickey).
fatal: Could not read from remote repository.

# Causes et solutions :
# 1. Clé SSH non configurée
$ ssh-keygen -t ed25519 -C "email@exemple.com"
$ cat ~/.ssh/id_ed25519.pub   # copier la clé publique
# Ajouter sur GitHub : Settings → SSH keys

# 2. Clé non chargée dans ssh-agent
$ eval "$(ssh-agent -s)" && ssh-add ~/.ssh/id_ed25519

# 3. Mauvaise URL (HTTPS au lieu de SSH)
$ git remote set-url origin git@github.com:user/repo.git

# Test de connexion SSH
$ ssh -T git@github.com
# Réponse attendue : "Hi username! You've successfully authenticated"
```

Erreur : fatal: refusing to merge unrelated histories

```
$ git pull
fatal: refusing to merge unrelated histories

# Cause : Les deux dépôts n'ont pas d'historique commun
# (par ex: init local puis push vers repo GitHub déjà existant)
# Solution :
$ git pull origin main --allow-unrelated-histories
$ git push
```

Erreur : Your local changes would be overwritten by merge

```
$ git pull
error: Your local changes to the following files would be
overwritten by merge: fichier.py

# Cause : Des modifications non commitées empêchent le pull
```

```
# Solutions :
# Option 1 : Stasher les modifications
$ git stash
$ git pull
$ git stash pop

# Option 2 : Commiter d'abord
$ git add . && git commit -m "WIP: sauvegarde"
$ git pull
```

10.3 Commandes avancées utiles

```
# git bisect — Trouver le commit qui a introduit un bug
$ git bisect start
$ git bisect bad              # le commit actuel est bogué
$ git bisect good v1.0.0    # v1.0.0 était bon
# Git checkout automatiquement à mi-chemin
$ python test.py && git bisect good || git bisect bad
# Répéter jusqu'à isolation du commit
$ git bisect reset            # terminer

# git worktree — Travailler sur plusieurs branches simultanément
$ git worktree add ../projet-hotfix hotfix/bug-123
$ cd ../projet-hotfix         # branche hotfix dans un autre répertoire
# Votre branche principale est toujours disponible dans le dossier original

# git notes — Ajouter des notes à un commit sans le modifier
$ git notes add -m "Testé avec Python 3.12 le 2024-03-15" HEAD
$ git log --notes

# git archive — Exporter le code sans l'historique Git
$ git archive --format=zip HEAD > mon-projet.zip
$ git archive --format=tar.gz v1.0.0 > release-1.0.0.tar.gz

# git shortlog — Résumé des commits par auteur
$ git shortlog -sn --all

# git bundle — Transférer un dépôt sans accès réseau
$ git bundle create repo.bundle main
$ git clone repo.bundle mon-projet-local

# git rerere — Mémoriser les résolutions de conflits
$ git config rerere.enabled true
# Désormais Git mémorise comment vous résolvez les conflits
# et les résout automatiquement si la même situation se reproduit
```

10.4 Configuration Git recommandée

```
# ~/.gitconfig — Configuration recommandée

[user]
    name = Votre Nom
    email = vous@exemple.com
```

```
    signingkey = VOTRE_CLE_GPG   # optionnel

[core]
    editor = code --wait         # VS Code comme éditeur
    autocrlf = input             # Linux/Mac : input, Windows : true
    whitespace = fix,-indent-with-non-tab,trailing-space,cr-at-eol

[init]
    defaultBranch = main

[pull]
    rebase = true                # git pull toujours avec rebase

[fetch]
    prune = true                 # supprimer auto les branches distantes
supprimées

[merge]
    conflictstyle = diff3        # afficher aussi la base dans les conflits

[diff]
    colorMoved = zebra

[alias]
    lg = log --oneline --graph --all --decorate --color
    st = status -s
    co = checkout
    sw = switch
    br = branch -a
    last = log -1 HEAD --stat
    undo = reset HEAD~1 --mixed
    unstage = restore --staged
    aliases = config --get-regexp alias

[credential]
    helper = osxkeychain         # macOS (ou 'manager' pour Windows)
```

10.5 Glossaire complet

Terme	Définition
Blob	Objet Git stockant le contenu brut d'un fichier (sans nom)
Branch	Pointeur mobile vers un commit ; permet le développement parallèle
Cherry-pick	Appliquer les modifications d'un commit spécifique sur une autre branche
CI/CD	Intégration Continue / Déploiement Continu — automatisation du pipeline
Clone	Copie complète d'un dépôt distant en local (historique inclus)
Commit	Instantané (snapshot) de l'état du projet à un moment donné
Conflict	Situation où Git ne peut pas fusionner automatiquement deux modifications
CNAME	Enregistrement DNS ou fichier indiquant un alias de domaine

Detached HEAD	État où HEAD pointe directement sur un commit plutôt que sur une branche
Fast-forward	Merge sans commit de merge quand l'historique est linéaire
Fetch	Télécharger les données distantes SANS modifier les branches locales
Fork	Copie d'un dépôt dans votre compte GitHub pour y travailler librement
HEAD	Pointeur vers la position courante dans l'historique
Hook	Script Git exécuté automatiquement avant/après certaines opérations
Index	Fichier .git/index représentant la zone de staging (Staging Area)
Issue	Ticket de suivi de bug, feature ou tâche sur GitHub
Merge	Intégration des modifications d'une branche dans une autre
Milestone	Regroupement d'issues/PRs autour d'un objectif ou d'une version
Origin	Alias conventionnel du dépôt distant depuis lequel on a cloné
PAT	Personal Access Token — token d'authentification HTTPS pour GitHub API
PR	Pull Request — demande d'intégration de modifications (GitHub)
Rebase	Rejouer des commits sur un nouveau parent (réécriture d'historique)
Ref	Référence vers un commit (branche, tag, HEAD, etc.)
Reflog	Journal local de tous les déplacements de HEAD
Release	Version publiée d'un projet, associée à un tag Git
Remote	Dépôt Git hébergé sur un serveur distant
Repository	Base de données Git contenant l'historique complet d'un projet
Runner	Machine virtuelle exécutant les workflows GitHub Actions
Secret	Variable chiffrée dans GitHub pour les tokens/mots de passe
SHA	Identifiant cryptographique unique de chaque objet Git (40 hex)
Snapshot	Copie complète de l'état des fichiers à un instant donné
Squash	Condenser plusieurs commits en un seul
SSH	Protocole sécurisé d'authentification par paire de clés
Staging Area	Zone intermédiaire entre le Working Directory et le commit
Stash	Sauvegarde temporaire de modifications non commitées
Tag	Étiquette permanente pointant vers un commit (ex: v1.0.0)
Tree	Objet Git représentant un répertoire et son contenu
Upstream	Dépôt original dont votre repo est un fork
Workflow	Fichier YAML GitHub Actions décrivant une automatisation
Working Dir	Répertoire de travail contenant les fichiers du projet

Cheat Sheet Git & GitHub

📋 **Référence 2 pages —** Section de référence rapide — à imprimer ou garder sous la main

⚡ Commandes du quotidien

```
# INITIALISATION
git init                          # Nouveau dépôt local
git clone URL                     # Cloner un dépôt distant
git clone git@github.com:u/r.git  # Cloner via SSH

# ÉTAT
git status                        # État des fichiers
git status -s                     # Format court
git log --oneline --graph --all   # Visualiser l'historique

# STAGING & COMMIT
git add fichier.py                # Stager un fichier
git add .                         # Stager tout
git add -p                        # Stager par morceaux (puissant !)
git commit -m "type: message"     # Commiter
git commit -am "fix: ..."         # Stager+commiter (fichiers trackés)
git commit --amend                # Modifier le dernier commit

# BRANCHES
git branch -a                     # Lister toutes les branches
git switch -c feature/nom         # Créer et aller sur une branche
git switch main                   # Changer de branche
git branch -d feature/nom         # Supprimer une branche
git merge feature/nom             # Merger dans la branche courante
git rebase main                   # Rebaser sur main

# REMOTE
git remote -v                     # Lister les remotes
git fetch --all --prune           # Télécharger (sans modifier local)
git pull --rebase                 # Télécharger + rebaser
git push -u origin feature/nom    # Pousser + set upstream
git push --force-with-lease       # Force push sécurisé

# ANNULER & CORRIGER
git restore fichier.py            # Annuler modifs WD
git restore --staged fichier.py   # Désindexer (unstage)
git reset --soft HEAD~1           # Défaire commit (garde staging)
git reset --mixed HEAD~1          # Défaire commit (garde WD)
git reset --hard HEAD~1           # ANNULER TOUT ⚠️
git revert HEAD                   # Annuler avec nouveau commit (safe)
git reflog                        # Trouver les commits perdus

# UTILITAIRES
git stash                         # Mettre de côté WIP
```

```
git stash pop                          # Récupérer WIP
git tag -a v1.0.0 -m "Stable"        # Créer un tag annoté
git diff --staged                      # Voir les modifs stagées
git blame fichier.py                   # Auteur de chaque ligne
```

⚡ GitHub Actions YAML minimal

```
name: CI
on:
  push:
    branches: ["main"]
  pull_request:
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - run: pip install -r requirements.txt
      - run: pytest
```

⚡ Workflows en 5 lignes

Scénario	Commandes
Nouvelle feature	git switch -c feature/x → code → git add -p → git commit → git push → PR
Corriger un bug urgent	git switch -c hotfix/bug main → fix → commit → push → PR → merge → tag
Contribution OS	Fork → clone → git remote add upstream → branche → code → rebase → PR
Annuler dernier commit	git reset --soft HEAD~1 (garde les modifs) ou git revert HEAD (safe)
Récupérer après reset --hard	git reflog → git reset --hard HEAD@{N} (ou git branch save <hash>)
Synchroniser fork	git fetch upstream → git rebase upstream/main → git push

⚡ Conventions de commits

Préfixe	Usage
`feat:`	Nouvelle fonctionnalité
`fix:`	Correction de bug
`docs:`	Documentation uniquement

`style:`	Formatage, espaces (pas de logique)
`refactor:`	Refactoring sans bug ni feature
`test:`	Ajout ou modification de tests
`chore:`	Dépendances, CI, configuration
`perf:`	Amélioration de performance
`ci:`	Configuration CI/CD

⚡ Règles d'or

- Ne jamais force-pusher sur une branche partagée (utiliser --force-with-lease)
- Ne jamais rebase des commits déjà pushés sur une branche partagée
- Ne jamais commiter de secrets (API keys, mots de passe) — même en privé
- Toujours utiliser git fetch avant git pull pour voir ce qui arrive
- Les messages de commit doivent expliquer POURQUOI, pas quoi (le code dit quoi)
- En cas de doute : git reflog — vos commits ne disparaissent jamais immédiatement