LE GUIDE COMPLET DU VERSIONNEMENT AVEC GIT

Le résumé de ce qu'il y a à savoir sur Git

Auteur: Marie Roger fokou K.

yaoundé, May 2025

This page intentionally left blank.

LE GUIDE COMPLET DU VERSIONNEMENT AVEC GIT

Le résumé de ce qu'il y a à savoir sur Git

Auteur: Marie Roger fokou K.

yaoundé, May 2025

This page intentionally left blank.

LE GUIDE COMPLET DU VERSIONNEMENT AVEC GIT

This page intentionally left blank.

Contents

This page intentionally left blank.

1 Introduction à Git

Auteur: Mariefk
Official Repository: GitHub Repository

Dans le monde du développement logiciel, il ne suffit plus d'écrire du code : il faut aussi savoir le gérer, le partager et le maintenir efficacement. C'est là qu'intervient Git, un système de contrôle de version devenu incontournable, et GitHub, la plateforme collaborative qui a révolutionné la façon dont les développeurs travaillent ensemble.

1.1 A qui s'adresse cet ouvrage?

Ce livre est conçu pour :

- Les débutants qui découvrent Git pour la première fois.
- Les étudiants ou développeurs en reconversion.
- Les développeurs confirmés souhaitant structurer leurs projets collaboratifs.
- Toute personne voulant publier un site web ou collaborer efficacement avec une équipe.

1.2 Ce que vous allez apprendre

Au fil des chapitres, vous allez :

- Comprendre les concepts fondamentaux de Git.
- Apprendre à utiliser Git en ligne de commande.
- Collaborer sur des projets via GitHub.
- Déployer des sites avec GitHub Pages.
- Mettre en place des workflows professionnels avec Git Flow.
- Découvrir des astuces, bonnes pratiques et outils avancés.

> *N'hesitez pas a suivre des tutoriels Youtube sur la chaine FreecodeCamp pour plus de clarté.*

2

Git – Les bases du contrôle de version

2.1 Qu'est-ce que Git ?

Git est un système de contrôle de version décentralisé (Distributed Version Control System – DVCS). Il permet de :

- Suivre l'historique des modifications d'un projet,
- Travailler à plusieurs sans écraser le code des autres,
- Créer des versions (branches), tester, fusionner, etc.

Git a été créé en 2005 par Linus Torvalds, le créateur de Linux, pour gérer le code source du noyau Linux.

Pourquoi utiliser Git ?

- Historique complet du code
- Travail en équipe facilité
- Sécurité : tout est stocké localement, rien n'est perdu
- Expérimentation sans risque : via les branches

2.2 Installer Git

Windows

1. Va sur `https://git-scm.com`
2. Clique sur **Download for Windows**
3. Lance l'installeur, choisis les options par défaut
4. Une fois installé, ouvre le Git Bash ou ton terminal favori

macOS

Dans le terminal, tape :

```
git --version
```

Si Git n'est pas installé, macOS proposera de l'installer automatiquement. Sinon, utilise :

```
brew install git
```

Linux (Ubuntu/Debian)

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

2.3 Créer un dépôt Git local

Un dépôt (*repository*) est un dossier dans lequel Git va suivre les fichiers.

Étapes :

```
mkdir mon-projet
cd mon-projet
git init
```

Résultat : un sous-dossier `.git` est créé, contenant tout ce que Git utilise pour suivre ton code.

2.4 Ajouter, valider, annuler

git status

Montre l'état actuel du dépôt :

```
git status
```

git add

Ajoute des fichiers à l'index (zone de préparation) :

```
git add fichier.txt      # Ajoute un fichier
git add .                # Ajoute tous les fichiers
```

git commit

Valide les changements ajoutés :

```
git commit -m "Ajout du fichier principal"
```

git log

Affiche l'historique des commits :

```
git log
```

Annuler un fichier ajouté

```
git restore --staged fichier.txt
```

2.5 Naviguer dans l'historique

Voir les commits :

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

Revenir à un ancien état (sans perdre les changements) :

```
git checkout <hash_du_commit>
```

Annuler un commit :

```
git revert <hash_du_commit>
```

À noter : `git reset` est plus radical, car il modifie l'historique (à utiliser avec prudence).

2.6 Ignorer des fichiers avec `.gitignore`

Certains fichiers ne doivent jamais être versionnés :

- Fichiers temporaires
- Clés secrètes
- Dossiers `node_modules`, `venv`, etc.

Crée un fichier `.gitignore` à la racine de ton dépôt, exemple :

```
# .gitignore
node_modules/
.env
*.log
.DS_Store
```

Ensuite, Git ne suivra plus ces fichiers.

□□ Note utile

Même si Git peut paraître complexe au début, sa maîtrise devient vite un atout incontournable. Pratique régulièrement, expérimente avec des dépôts test, et n'hésite pas à consulter l'aide intégrée :

```
git help <commande>
```

ou encore :

```
git status
```

pour rester toujours au courant de l'état de ton projet !

3

Branches, fusions et conflits

1. Travailler avec les branches (branch, checkout, switch)

Qu'est-ce qu'une branche ?

Une **branche** (*branch*) est une version parallèle de ton code. Elle permet de :

- Travailler sur une nouvelle fonctionnalité
- Corriger un bug
- Expérimenter sans toucher à la version principale (`main` ou `master`)

> **Info**
>
> Git crée automatiquement une branche appelée `main` lors de l'initialisation d'un dépôt.

Créer une branche

```
git branch nom-de-ta-branche
```

Changer de branche (2 méthodes)

```
# Ancienne méthode
git checkout nom-de-ta-branche

# Nouvelle méthode (recommandée)
git switch nom-de-ta-branche
```

Créer et se déplacer en même temps

```
git switch -c nouvelle-branche
```

Liste des branches

```
git branch
```

La branche actuelle est indiquée par un *.

2. Fusionner du code (merge)

Pour fusionner une branche de travail dans `main`, fais :

```
git switch main           # Se placer sur la branche cible
git merge feature-login   # Fusionner la branche source
```

Conseil

Toujours committer avant de merger.

3. Résoudre les conflits

Quand se produit un conflit ?

Un conflit apparaît quand deux branches modifient la même ligne ou fichier. Exemple :

```
<<<<<<< HEAD
Contenu depuis main
=======
Contenu depuis feature
>>>>>>> feature
```

Étapes pour résoudre un conflit

- Ouvre le fichier concerné
- Modifie manuellement le contenu
- Ajoute le fichier corrigé :

  ```
  git add nom-du-fichier
  ```

- Termine la fusion :

  ```
  git commit
  ```

Astuce

Des éditeurs comme VS Code facilitent la gestion des conflits avec une interface visuelle.

4. Rebaser proprement (rebase)

C'est quoi un rebase ?

Le **rebase** rejoue les commits d'une branche sur une autre, pour garder un historique linéaire :

```
git switch feature
git rebase main
```

> **Attention**
>
> Ne fais pas de `rebase` sur des branches déjà partagées sur GitHub.

Forcer la mise à jour après un rebase

```
git push --force
```

5. Bonnes pratiques avec les branches

- Crée une branche pour chaque fonctionnalité : `feature/auth`, `fix/bug-navbar`, `hotfix/login-crash`
- Utilise des préfixes :
  - `feature/` pour les nouvelles fonctionnalités
  - `fix/` ou `bugfix/` pour les corrections
  - `hotfix/` pour les urgences
  - `chore/` pour la maintenance
- Ne travaille pas directement sur `main`
- Mets à jour proprement avec :

  ```
  git pull --rebase
  ```

- Supprime les branches fusionnées :

  ```
  git branch -d feature-login
  ```

4

GitHub – Collaboration en ligne

4.1 Créer un compte GitHub

Qu'est-ce que GitHub ?

GitHub est une plateforme web qui permet :

- d'héberger du code Git en ligne,
- de collaborer facilement en équipe,
- de suivre les bugs et les tâches,
- de gérer les versions et déploiements.

GitHub repose entièrement sur Git.

Création du compte

1. Va sur `https://github.com`
2. Clique sur **Sign up**
3. Crée un nom d'utilisateur, mot de passe, e-mail
4. Confirme ton compte par e-mail

4.2 Lier un dépôt local à GitHub

Tu peux relier un dépôt Git local à un dépôt GitHub distant pour synchroniser ton code.

Étapes

1. Créer un nouveau dépôt sur GitHub (sans `README` ni `.gitignore`)
2. Dans ton dépôt local, exécute :

```
git remote add origin https://github.com/ton-user/nom-du-repo.git
```

3. Pousser ton code vers GitHub :

```
git push -u origin main
```

Pour récupérer les changements

```
git pull origin main
```

Tu peux remplacer `main` par le nom de ta branche si nécessaire.

Vérifier l'origine du dépôt distant

```
git remote -v
```

4.3 Travailler avec plusieurs contributeurs

Lorsque plusieurs développeurs collaborent sur le même projet, GitHub devient essentiel pour :

- Partager les branches,
- Ouvrir des discussions (*Issues*),
- Valider du code (*Pull Requests*),
- Suivre qui a modifié quoi.

Bonne pratique

1. Chaque développeur clone le dépôt distant
2. Crée sa propre branche :

```
git switch -c feature/ajout-commentaires
```

3. Pousse sa branche :

```
git push origin feature/ajout-commentaires
```

4.4 Les Pull Requests et le Code Review

Qu'est-ce qu'une Pull Request (PR) ?

Une *Pull Request* (*PR*) est une demande de fusion d'une branche vers une autre (souvent vers `main`), sur GitHub.

Avantages

- Permet de relire le code avant de l'intégrer
- Démarre une discussion technique
- Permet des *reviews* collaboratives

Étapes

1. Pousser ta branche sur GitHub
2. Ouvrir une *Pull Request*
3. Ajouter une description claire
4. Les coéquipiers commentent, approuvent, demandent des changements
5. Une fois validée : fusion de la branche

Tu peux squasher les commits ou faire un rebase avant de fusionner pour garder un historique propre.

4.5 Gérer les Issues et les Projets

Les Issues (problèmes)

Les *Issues* permettent de :

- Signaler des bugs
- Demander de nouvelles fonctionnalités
- Suivre les tâches

Une *Issue* contient généralement :

- Un titre et une description
- Des commentaires
- Des assignations
- Des labels (ex : `bug`, `enhancement`, `urgent`)

Les Projets GitHub

Fonctionnent comme un tableau *Kanban* :

- Crée des colonnes : `À faire`, `En cours`, `Terminé`
- Ajoute-y des *Issues* ou des *Pull Requests*
- Utilise-les pour organiser le travail en équipe
- Les projets peuvent être gérés en mode classique ou via `GitHub Projects (Beta)` avec plus d'automatisation

4.6 Utiliser GitHub Desktop (optionnel)

GitHub Desktop est une application graphique (*GUI*) qui simplifie l'utilisation de Git et GitHub.

Idéal pour :

- Les débutants
- Les utilisateurs non à l'aise avec le terminal

Installation

- Télécharger sur `https://desktop.github.com`

Fonctions

- Cloner des dépôts
- Visualiser les commits, branches, conflits
- Pousser, tirer, fusionner sans ligne de commande

Cela reste utile de comprendre Git en ligne de commande avant de tout faire en mode graphique.

5

Déploiement avec GitHub Pages

5.1 Qu'est-ce que GitHub Pages ?

GitHub Pages est un service gratuit de GitHub permettant de déployer des sites web statiques directement depuis un dépôt GitHub.

Ce que tu peux héberger

- Des sites HTML/CSS/JS simples
- Des documentations
- Des projets React, Vue, etc.
- Des portfolios, blogs, etc.

Avantages

- Gratuit
- Rapide à mettre en place
- Accessible via une URL comme :
  `https://nom-utilisateur.github.io/nom-du-repo/`

5.2 Déployer un site statique (HTML/CSS/JS)

Étapes

1. Créer un dépôt GitHub
2. Ajouter ton projet HTML/CSS/JS dans ce dépôt
3. Aller dans `Settings > Pages`
4. Sélectionner la branche (`main`) et le dossier `/root` ou `/docs`
5. GitHub génère une URL du type :
   `https://nom-utilisateur.github.io/nom-du-repo/`

Pour que GitHub détecte le site, un fichier `index.html` *doit être à la racine (ou dans le dossier* `docs`*).*

5.3 Déployer un projet React avec Vite

Déployer un projet React avec Vite nécessite quelques ajustements.

Étapes complètes

a. Modifier `vite.config.js`

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  base: '/nom-du-repo/',  // <- très important !
  plugins: [react()],
})
```

b. Ajouter un script de déploiement

- Installer `gh-pages` :

  ```
  npm install --save-dev gh-pages
  ```

- Ajouter ces scripts dans `package.json` :

  ```
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "predeploy": "npm run build",
    "deploy": "gh-pages -d dist"
  }
  ```

c. Initialiser Git et pousser vers GitHub

```
git init
git add .
git commit -m "Initial commit"
git branch -M main
git remote add origin https://github.com/ton-utilisateur/nom-du-repo.git
git push -u origin main
```

d. Lancer le déploiement

```
npm run deploy
```

Ton site sera alors disponible à :
https://ton-utilisateur.github.io/nom-du-repo/

5.4 Résoudre les erreurs courantes

Erreur	Solution
Page blanche	Vérifie le champ `base` dans `vite.config.js`
404 sur les routes	`React Router` doit être configuré avec `BrowserRouter` et `basename`
Permission denied	Vérifie tes droits Git sur le dépôt
`gh-pages` non trouvé	Vérifie que le package est bien installé dans les `devDependencies`

Regarde la console navigateur pour les erreurs réseau.

5.5 Utiliser un nom de domaine personnalisé

Tu peux relier ton site GitHub Pages à un domaine (ou sous-domaine) comme `www.monsite.com`.

Étapes

1. Acheter un nom de domaine chez un registrar (OVH, GoDaddy, etc.)
2. Créer un fichier `CNAME` dans ton projet contenant :

   ```
   www.monsite.com
   ```

3. Dans GitHub, aller dans : `Settings > Pages > Custom domain` et indiquer ton domaine
4. Configurer un enregistrement `CNAME` dans le DNS :

   ```
   www.monsite.com → nom-utilisateur.github.io
   ```

□ *Cela peut prendre quelques heures à se propager.*

6

Git Flow et stratégies de branche

6.1 Pourquoi utiliser Git Flow ?

Le problème

Lorsque plusieurs développeurs travaillent sur le même projet sans structure claire de branches, cela peut vite devenir un chaos :

- Conflits fréquents,
- Fusions (*merges*) désorganisées,
- Bugs en production.

La solution

Git Flow propose un workflow structuré basé sur différentes branches avec des rôles bien définis :

- Développement,
- Préparation à la livraison,
- Maintenance de la production,
- Intégration de nouvelles fonctionnalités.

Cela améliore :

- la clarté du cycle de développement,
- la collaboration,
- la gestion des versions.

6.2 Présentation du workflow Git Flow

Voici les branches principales utilisées dans Git Flow :

Branche	Rôle
`main` (ou `master`)	Contient le code en production
`develop`	Contient le code en cours de développement
`feature/*`	Pour le développement de nouvelles fonctionnalités
`release/*`	Prépare une version stable avant livraison
`hotfix/*`	Corrige un bug critique en production

Cycle typique

1. Tu crées une branche `feature` pour une nouvelle fonction : `feature/ajout-commentaires`
2. Tu merges cette branche dans `develop`
3. Quand toutes les features sont prêtes, tu crées une branche `release`
4. Une fois testée, tu merges la `release` dans :
   - `main` (pour la mise en production)
   - `develop` (pour conserver les changements)
5. En cas d'urgence, tu crées une branche `hotfix` depuis `main`

6.3 Installation et commandes principales

Installation

- Sous Linux/Mac :

  ```
  brew install git-flow
  ```

  ou :

  ```
  sudo apt install git-flow
  ```

- Sous Windows :
  Télécharger via Git SCM ou utiliser Git Bash pour les commandes.

Initialisation dans un projet

```
git flow init
```

Tu choisis les noms de branches (main, develop, etc.).

Commandes courantes

Objectif	Commande
Démarrer une feature	`git flow feature start nom`
Terminer une feature	`git flow feature finish nom`
Démarrer une release	`git flow release start v1.0`
Terminer une release	`git flow release finish v1.0`
Démarrer un hotfix	`git flow hotfix start correctif-urgent`
Terminer un hotfix	`git flow hotfix finish correctif-urgent`

Chaque commande crée les branches, effectue les fusions et pousse les changements si demandé.

6.4 Alternatives modernes à Git Flow

GitHub Flow

- Branche `main` uniquement
- Chaque fonctionnalité a sa propre branche
- Merge via Pull Request après revue
- Déploiement fréquent et automatisé

Trunk-Based Development

- Tout le monde travaille sur une seule branche principale (`main`)
- Commits petits et fréquents
- Tests automatisés
- Très utilisé en CI/CD

GitLab Flow

- Mixe Git Flow et GitHub Flow
- Intègre la gestion d'environnements (staging, production, etc.)

6.5 Choisir la bonne stratégie pour ton projet

Critère	**Recommandation**
Projet solo ou petit projet	GitHub Flow (simple, rapide)
Équipe de 2-5 développeurs avec livraisons fréquentes	GitHub Flow + Pull Requests
Équipe structurée avec des cycles de release clairs	Git Flow
Environnement DevOps/CI/CD	Trunk-Based Development
Besoin de validation rigoureuse avant mise en production	Git Flow ou GitLab Flow

Il n'y a pas de "bonne" ou "mauvaise" stratégie universelle. Choisis selon ton contexte d'équipe, la fréquence de livraison et ton niveau de maturité DevOps.

7 Bonnes pratiques Git au quotidien

Utiliser Git efficacement, c'est aussi adopter de bonnes habitudes. Voici les principales bonnes pratiques à suivre.

7.1 Des messages de commit clairs

Utilise des messages simples, descriptifs et standardisés :

```
feat: ajout du système de commentaires
fix: correction du bug de connexion
refactor: simplification du code du formulaire
```

Convention recommandée : **Conventional Commits**

7.2 Commits atomiques

Chaque commit doit :

- Réaliser une seule tâche claire,
- Pouvoir être testé ou annulé indépendamment.

7.3 Toujours tester avant de commit

N'ajoute jamais de code cassé sur les branches `main` ou `develop`. Utilise des branches de fonctionnalité pour expérimenter et tester.

7.4 Utiliser `.gitignore`

Évite de versionner les fichiers qui ne doivent pas être suivis par Git, comme :

- Fichiers de configuration personnelle : `.env`
- Dépendances : `node_modules/`
- Fichiers générés : `dist/`, `build/`, `.DS_Store`

7.5 Commandes Git avancées

`git stash`

Sauvegarde temporairement tes modifications sans les committer :

```
git stash
git stash list
git stash pop
```

`git tag`

Crée une étiquette pour marquer une version spécifique :

```
git tag v1.0
git push origin v1.0
```

□ **`git reset`**

Annule un ou plusieurs commits (□ à utiliser avec précaution) :

```
git reset --soft HEAD~1     # Garde les modifications
git reset --hard HEAD~1     # Supprime tout
```

`git cherry-pick`

Applique un commit spécifique d'une branche vers une autre :

```
git cherry-pick abc1234
```

7.6 Git dans VS Code

VS Code propose une intégration Git puissante :

Fonctions intégrées

- **Source Control** : voir les fichiers modifiés
- Commit, Push, Pull en un clic
- Résolution de conflits avec interface graphique
- Historique des commits

Extensions utiles

- **GitLens** : historique, blame, navigateurs de branches
- **Git Graph** : visualisation graphique du repo
- **GitHub Pull Requests** : gérer les PR sans quitter VS Code

7.7 Gérer les clés SSH pour GitHub

Pourquoi ?

Permet une authentification sécurisée sans entrer ton mot de passe à chaque `git push`.

Étapes

1. Génère une clé :

   ```
   ssh-keygen -t ed25519 -C "ton@email.com"
   ```

2. Copie la clé publique :

   ```
   cat ~/.ssh/id_ed25519.pub
   ```

3. Va sur GitHub → `Settings → SSH and GPG keys → New SSH key`
4. Ajoute la clé
5. Configure Git pour utiliser SSH :

   ```
   git remote set-url origin git@github.com:utilisateur/repo.git
   ```

7.8 Scripts d'automatisation avec Git Hooks

Les Git Hooks sont des scripts qui s'exécutent automatiquement à certains moments du cycle Git.

Où les trouver ?

Dans le dossier `.git/hooks`

Exemples courants

- `pre-commit` : lancer des tests ou un linter avant chaque commit
- `commit-msg` : vérifier le format des messages de commit
- `post-merge` : exécuter un `npm install` après un merge

Exemple de hook `pre-commit`

```
#!/bin/sh
npm run lint
```

Rends-le exécutable :

```
chmod +x .git/hooks/pre-commit
```

Tu peux aussi utiliser des outils comme **Husky** pour gérer facilement les hooks dans les projets JavaScript modernes.

8 Annexes

8.1 Glossaire Git / GitHub

Terme	Définition
Commit	Enregistrement d'un ensemble de modifications dans un dépôt Git
Branche	Ligne parallèle de développement dans un projet
Merge	Fusionner le contenu d'une branche dans une autre
Pull Request (PR)	Demande de fusion d'une branche sur GitHub, avec possibilité de relecture
Clone	Copier un dépôt Git distant sur sa machine locale
Remote	Nom d'un dépôt distant, souvent `origin`
Rebase	Réappliquer des commits sur une autre base (alternative à merge)
Stash	Sauvegarde temporaire de modifications non validées
Tag	Étiquette utilisée pour marquer une version stable ou importante
HEAD	Pointeur vers le dernier commit actuellement vérifié

8.2 Fiches mémo de commandes

Commandes Git de base

- `git init` : Initialiser un dépôt local
- `git status` : Voir l'état des fichiers
- `git add .` : Ajouter tous les fichiers au staging
- `git commit -m "message"` : Créer un commit
- `git log` : Voir l'historique
- `git diff` : Voir les modifications
- `git reset` : Annuler des changements

Branches et fusion

- `git branch` : Lister les branches

- `git branch nom` : Créer une branche
- `git checkout nom` : Changer de branche
- `git merge branche` : Fusionner une branche
- `git rebase branche` : Rebaser sur une autre branche

□ GitHub et distant

- `git remote add origin URL` : Ajouter un dépôt distant
- `git push -u origin main` : Envoyer le code sur GitHub
- `git pull` : Récupérer les modifications
- `git clone URL` : Cloner un projet GitHub

8.3 Liens utiles

- **Site officiel de Git** → https://git-scm.com
- **Documentation GitHub** → https://docs.github.com
- **Git Flow cheatsheet** → https://danielkummer.github.io/git-flow-cheatsheet/
- **Learn Git Branching (interactive)** → https://learngitbranching.js.org/
- **Oh My Git! (jeu éducatif)** → https://ohmygit.org/
- **Pro Git Book (gratuit)** → https://git-scm.com/book/fr/v2

8.4 Ressources complémentaires

Vidéos YouTube

- Grafikart – Tutoriels Git en français
- Traversy Media – Git GitHub crash course

Livres recommandés

- **Pro Git** par Scott Chacon (en ligne gratuitement)
- **Git Pocket Guide** par Richard E. Silverman

Extensions VS Code

- GitLens
- GitHub Pull Requests and Issues
- Git Graph

This page intentionally left blank.

This page intentionally left blank.