

Développement et contrôle de logiciel



Rédacteur : HAJIBA IFRAH

Gestion de version avec GIT



Problématiques

Perte de données : Écrasement accidentel des fichiers.

Difficulté à suivre les modifications : Pas de suivi clair des évolutions du projet.

Collaboration chaotique : Plusieurs personnes modifiant un fichier en même temps sans coordination.

Absence d'historique : Impossible de revenir à une version précédente.

Risques accrus d'erreurs : Mauvaises manipulations sans possibilité de correction facile.



Besoins

Suivi des modifications : Chaque modification est historiée.

Travail collaboratif facilité : Plusieurs développeurs peuvent travailler sur le même projet.

Possibilité de revenir en arrière : On peut restaurer une version précédente en cas d'erreur.

Meilleure organisation : Structuration des versions via des branches.

Amélioration de la qualité du code : Revue de code facilitée par l'historique et les branches dédiées.



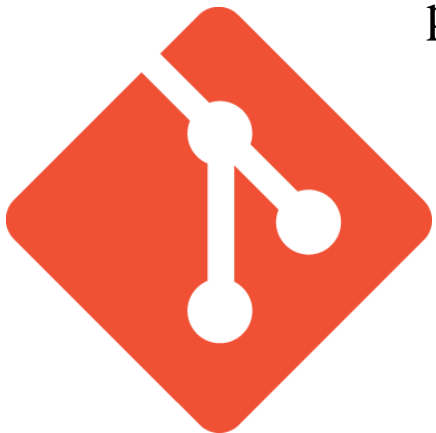
Introduction à la Gestion de Version

Qu'est-ce que la gestion de version ?

La gestion de version est un système qui permet de suivre les modifications apportées à un fichier ou un ensemble de fichiers.

Elle est essentielle dans le développement logiciel pour conserver un historique des changements et permettre le travail collaboratif.

Exemple : Sauvegarder différentes versions d'un document pour revenir à une version précédente en cas d'erreur.



Types de systèmes de gestion de version

1. Système de gestion de version centralisé

Un serveur central stocke toutes les versions des fichiers.

Les utilisateurs obtiennent une copie de travail, mais doivent se connecter au serveur pour enregistrer les modifications.

Exemple : Subversion (SVN).

Avantages :

Contrôle centralisé facilitant la gestion.

Facile à comprendre pour les petites équipes.

Inconvénients :

Dépendance à un serveur central (si le serveur tombe, plus d'accès aux versions).

Risque de perte de données en cas de panne du serveur.



Types de systèmes de gestion de version

2. Système de gestion de version distribué

Chaque utilisateur possède une copie complète du dépôt avec tout l'historique.

Les modifications peuvent être faites localement et synchronisées plus tard avec d'autres dépôts.

Exemples : Git, Mercurial.

Avantages :

Pas de point de défaillance unique (chaque copie contient l'historique complet).

Travail hors ligne possible.

Fusion des modifications plus flexible.

Inconvénients :

Peut être plus complexe à prendre en main.

Consommation de stockage plus importante (chaque utilisateur a une copie complète du dépôt).



Types de systèmes de gestion de version

2. Système de gestion de version distribué

Chaque utilisateur possède une copie complète du dépôt avec tout l'historique.

Les modifications peuvent être faites localement et synchronisées plus tard avec d'autres dépôts.

Exemples : Git, Mercurial.

Avantages :

Pas de point de défaillance unique (chaque copie contient l'historique complet).

Travail hors ligne possible.

Fusion des modifications plus flexible.

Inconvénients :

Peut être plus complexe à prendre en main.

Consommation de stockage plus importante (chaque utilisateur a une copie complète du dépôt).



Types de systèmes de gestion de version

Comparaison entre VCS et DVCS

Critère	VCS (Centralisé)	DVCS (Distribué)
Serveur requis	Oui	Non (local possible)
Travail hors ligne	Non	Oui
Performance	Moins rapide	Plus rapide
Gestion des branches	Limitée	Puissante
Serveur requis	Oui	Non (local possible)



Introduction à Git

Git est un **système de gestion de versions** distribué permettant de suivre l'historique des modifications d'un projet.

Ce système est créé par Linus Torvalds en 2005.

Utilisé pour la collaboration, il permet à plusieurs développeurs de travailler simultanément.



Fonctionnalités Clés

Suivi des Versions

Git permet de suivre chaque modification effectuée sur les fichiers d'un projet sous forme de **commits**, qui représentent des instantanés de l'état du code à un moment donné.

Fonctionnement :

Chaque commit contient un identifiant unique (SHA), un message descriptif, une date, et les informations de l'auteur.

Les changements entre les **commits** peuvent être comparés et restaurés si nécessaire.



Les développeurs peuvent naviguer dans l'historique pour retrouver une version spécifique du projet.



Fonctionnalités Clés

Commandes principales :

bash

 Copier  Modifier

```
git add <fichier>      # Ajouter un fichier à la zone de staging
git commit -m "Message" # Valider les modifications avec un message
git log                 # Afficher l'historique des commits
git diff                 # Voir les différences entre versions
```



Fonctionnalités Clés

Commandes principales :

- **HEAD** est un pointeur vers le dernier commit de la branche active.
- **git checkout HEAD~1** : se déplacer vers le commit précédent.
- **git reset --soft HEAD~1** : annuler le dernier commit sans perdre les modifications.
- **git reset --hard HEAD~1** : annuler et supprimer complètement le dernier commit.



Fonctionnalités Clés

Suivi des Versions

Avantages :

- Historique clair des modifications.
- Revenir à une version antérieure en cas d'erreur.
- Traçabilité des contributions.



Fonctionnalités Clés

Développement en Branches

Les branches permettent de travailler en parallèle sur plusieurs fonctionnalités ou corrections sans affecter le code principal.

Fonctionnement :

- La branche principale (main ou master) représente la version stable du projet.
- Les développeurs créent des branches secondaires pour implémenter de nouvelles fonctionnalités, expérimenter, ou corriger des bugs.
- Une fois les modifications finalisées et testées, les branches secondaires peuvent être fusionnées avec la branche principale.





Fonctionnalités Clés

Développement en Branches

Commandes principales :

bash

 Copier  Modifier

```
git branch <nom>      # Créer une nouvelle branche  
git checkout <nom>     # Passer à une branche  
git branch             # Lister les branches disponibles
```



Fonctionnalités Clés

Développement en Branches

Avantages :

- Développement isolé pour chaque fonctionnalité.
- Facilite les contributions collaboratives.
- Prévention des bugs sur la branche principale.



Fonctionnalités Clés

Supprimer une branche

En local : `git branch -d nom_branche` (ou `-D` pour forcer la suppression).

Sur un dépôt distant : `git push origin --delete nom_branche`



Fonctionnalités Clés

Historique Complet

Git conserve un historique détaillé et permanent de toutes les modifications apportées au projet.

Fonctionnement :

Chaque commit est stocké avec ses métadonnées (auteur, date, message, modifications). L'historique peut être consulté sous différentes formes : graphique, simple liste, etc.



Fonctionnalités Clés

Historique Complet

Commandes principales :

bash

 Copier  Modifier

```
git log           # Historique détaillé  
git log --oneline # Affichage condensé  
git log --graph   # Vue graphique des branches
```



Fonctionnalités Clés

Historique Complet

Avantages :

- Transparence complète sur les changements apportés au projet.
- Diagnostic rapide des erreurs.
- Suivi de l'évolution du projet au fil du temps.



Fonctionnalités Clés

Fusion et Résolution de Conflits

La fusion permet d'intégrer les modifications d'une branche dans une autre. Cependant, des conflits peuvent survenir lorsque les mêmes lignes de code sont modifiées dans les deux branches.

Fonctionnement :

Git détecte automatiquement les conflits.

Les conflits doivent être résolus manuellement.

Une fois résolus, il faut valider les modifications fusionnées.




Fonctionnalités Clés

Fusion et Résolution de Conflits

Commandes principales :

bash

 Copier  Modifier

```
git merge <branche>      # Fusion d'une branche
git status                 # Vérification des conflits
git add <fichier>          # Ajout des résolutions de conflits
git commit                 # Validation après résolution
```



Fonctionnalités Clés

Fusion et Résolution de Conflits

Avantages :

- Permet la collaboration entre plusieurs développeurs.
- Conservation d'une version cohérente du code.
- Résolution fine des conflits.



Fonctionnalités Clés

Travail Local et Distant

Git permet de travailler à la fois localement sur votre machine et de synchroniser les changements avec un dépôt distant hébergé sur des plateformes comme GitHub, GitLab ou Bitbucket.

Fonctionnement :

- Les modifications sont effectuées localement et validées avec commit.
- Les changements peuvent être synchronisés avec le dépôt distant via push.
- Les mises à jour du dépôt distant sont récupérées avec pull.





Fonctionnalités Clés

Travail Local et Distant

Commandes principales :

bash

 Copier  Modifier

```
git remote add origin <url> # Associer un dépôt distant
git push                      # Envoyer les modifications
git pull                     # Récupérer les mises à jour
git clone <url>              # Cloner un dépôt distant
```



Fonctionnalités Clés

Travail Local et Distant

Avantages :

- **Marquage des Versions Importantes** : Identifier rapidement les versions stables ou milestones (v1.0).
- **Traçabilité** : Retrouver facilement l'état exact d'une version pour déploiement ou correction.
- **Documentation** : Les tags annotés permettent d'ajouter des messages descriptifs.
- **Isolation** : Facilite la création de branches à partir d'une version spécifique.
- **CI/CD** : Déclenchement automatisé de pipelines sur création de tags.
- **Partage** : Les tags sont synchronisables avec des dépôts distants (git push origin <tag>).
- **Versionnement Sémantique** : Pratique pour structurer les releases (v1.0.0).



Fonctionnalités Clés

Supprimer les modifications non enregistrées

git checkout -- nom_fichier : annule les modifications d'un fichier spécifique.

git reset --hard : annule toutes les modifications non commit.

git clean -df : supprime les fichiers non suivis.



Fonctionnalités Clés

SHA dans Git

SHA (Secure Hash Algorithm) est un identifiant unique de 40 caractères généré pour chaque commit.

Il garantit l'intégrité des données et permet de référencer précisément un commit.

Exemple : `git log --oneline` affiche les SHA des commits sous forme abrégée.



Fonctionnalités Clés

Git Tagging (Gestion des Versions)

Les **tags** marquent des points spécifiques dans l'historique, souvent pour signaler une version stable (release).

Deux types de tags :

- **Légers (lightweight)** : Simple pointeur vers un commit.
- **Annotés (annotated)** : Stockent des métadonnées (message, auteur, date).





Fonctionnalités Clés

Git Tagging (Gestion des Versions)

Commandes principales :

bash

 Copier  Modifier

```
git tag v1.0 # Créer un tag simple
git tag -a v1.0 -m "Version stable 1.0" # Tag annoté
git push origin v1.0 # Envoyer le tag distant
```



Fonctionnalités Clés

Envoyer un tag vers un dépôt distant

- **git tag -a v1.0 -m "Version 1.0"** : créer un tag annoté.
- **git push origin v1.0** : envoyer un tag spécifique.
- **git push origin --tags** : envoyer tous les tags locaux.



Fonctionnalités Clés

Revenir à une version précédente

git checkout commit_SHA : naviguer vers un commit précis en mode détaché.

git reset --hard commit_SHA : restaurer un commit en supprimant les modifications ultérieures.

git revert commit_SHA : créer un commit annulant les changements d'un commit précédent.



Fonctionnalités Clés

Afficher les dépôts distants associés

- **git remote -v** : affiche les dépôts distants liés.
- **git remote add origin URL** : ajoute un dépôt distant.
- **git remote remove origin** : supprime un dépôt distant.



Fonctionnalités Clés

Utilisation de git stash

git stash : stocke temporairement les modifications non commit.

git stash list : affiche la liste des stashes.

git stash pop : applique et supprime le dernier stash.

git stash apply stash@{n} : applique un stash spécifique.



Fonctionnalités Clés

Cherry-Picking dans Git

Le cherry-picking est une technique qui permet d'appliquer un commit spécifique d'une branche à une autre, sans fusionner toute la branche.

Utilisation de git cherry-pick

1. Identifier le commit à appliquer avec git log :

```
git log --oneline
```

Exemple de sortie :

```
a1b2c3d Correction d'un bug  
e4f5g6 Ajout d'une nouvelle fonctionnalité
```

2

2. Appliquer un commit spécifique :

```
git cherry-pick a1b2c3d
```

3. Gérer les conflits si nécessaire et finaliser avec :

```
git commit
```



Fonctionnalités Clés

Historique graphique

- **git log --graph --oneline --all** : affiche un historique en mode graphique simplifié.
- **gitk** : outil graphique pour visualiser l'historique.
- **git log --pretty=format:"%h - %an, %ar : %s"** : personnaliser l'affichage de l'historique.



Fonctionnalités Clés

Hooks dans Git

- Scripts exécutés automatiquement lors d'événements spécifiques (pre-commit, pre-push, etc.).
- Situés dans **.git/hooks/**, ils peuvent être personnalisés pour exécuter des tests, valider du code, etc.
- Exemple de hook pre-commit : empêcher le commit si du code non formaté est détecté.



Fonctionnalités Clés

Fichier .gitignore

Qu'est-ce que .gitignore ?

Le fichier .gitignore contient une liste de fichiers et dossiers que Git doit ignorer. Ces fichiers ne seront ni suivis ni inclus dans les commits.

Utilisation Principale :

Empêcher le suivi des fichiers sensibles, volumineux ou spécifiques à une machine locale.

•Exemples :

- Fichiers de configuration (config.json)
- Données temporaires (*.log, *.tmp)
- Dossiers spécifiques (node_modules, __pycache__)





Fonctionnalités Clés

Fichier .gitignore

Exemple de Fichier .gitignore

gitignore

 Copier  Modifier

```
# Ignorer les fichiers Python compilés
```

```
*.pyc
```

```
__pycache__/
```

```
# Ignorer les fichiers de logs
```

```
*.log
```

```
# Ignorer les dossiers de dépendances Node.js
```

```
node_modules/
```

```
# Ignorer les fichiers de configuration locaux
```

```
.env
```



Bonnes Pratiques

- Utiliser des branches pour chaque fonctionnalité (Feature Branches).
- Rédiger des messages de commit explicites.
- Faire des commits atomiques (un seul changement par commit).
- Tirer (pull) régulièrement pour éviter les conflits.
- Utiliser .gitignore pour exclure les fichiers inutiles du suivi.



Merci pour votre attention