

Développement Continu — Gestion de version (Git)

Workflow simple, historique propre, collaboration fluide

🧩 Série : Pilier de l'Agilité DevOps – Partie 2 (suite logique de la Présentation 1)

git rebase

Pourquoi la gestion de version ?



Traçabilité

Qui a modifié
Quoi a été changé
Quand et **pourquoi**



Collaboration

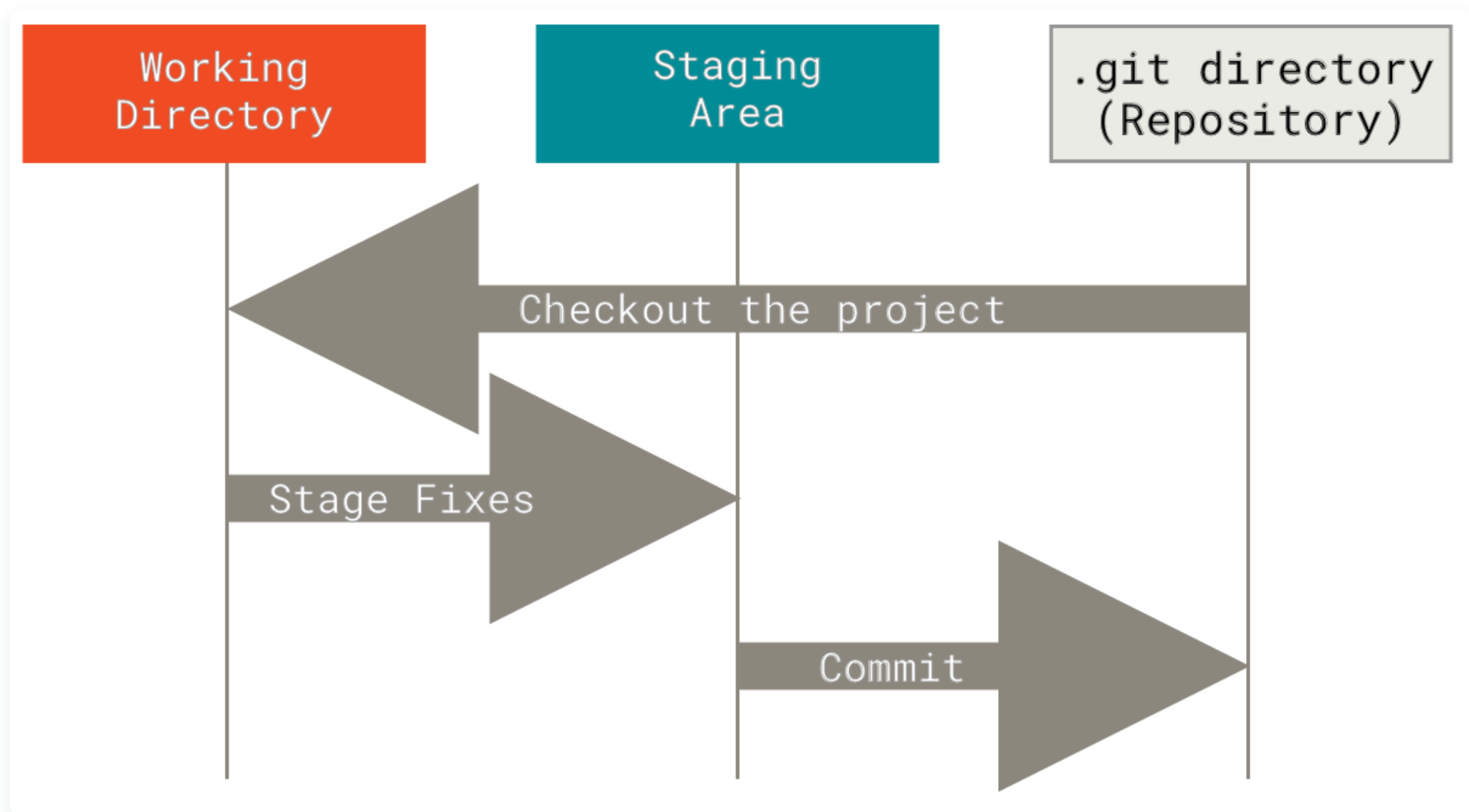
Branches isolées
PR/MR pour révision
Fusion sans conflit



Automatisation

Base pour **CI/CD**
Déploiement continu
Qualité assurée

Modèle mental Git (les 3 états)



Working Directory

Modifications locales

Fichiers que vous éditez
Changements non suivis



Staging Area

Sélection

Préparation du commit
Changements prêts à être enregistrés



Repository

Commits (historique)

Snapshots permanents
Traçabilité complète

Les commandes du quotidien



git status

Vérifie l'état des fichiers

```
$ git status
```



git add

Ajoute des fichiers au staging

```
$ git add fichier.txt  
$ git add .
```



git commit

Enregistre les changements

```
$ git commit -m  
"Message"
```



git diff

PPT

Affiche les différences

```
$ git diff  
$ git diff --staged
```



git log

Affiche l'historique

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



git switch

Bascule entre les branches

```
$ git switch main  
$ git switch -c  
nouvelle-branche
```



git pull

Récupère et fusionne les changements

```
$ git pull origin main
```



git push

Envoie les changements au dépôt distant

```
$ git push origin main
```



git restore

Restaure les fichiers

```
$ git restore  
fichier.txt
```

Routine journalière (Continuous Dev)

1



Partir propre

Synchroniser avec **main** pour éviter les conflits

```
$ git pull --ff-only origin main
```

2



Branche courte

Créer une branche **feature** avec un nom explicite

```
$ git switch -c feat/authentification
```

3



Commits atomiques

Petits commits **logiques** et **testables**

```
$ git add .  
$ git commit -m "feat: ajouter formulaire  
login"
```

4



Collaboration

Partager via **PR/MR** pour révision et fusion

```
$ git push origin feat/authentification  
$ # Créer PR/MR sur GitHub/GitLab
```

Stratégie de branches (simple & efficace)



Trunk-based Development

Petites features ($\leq 2-3$ jours) directement sur des branches courtes depuis **main**



main protégée

- **Review** obligatoire avant fusion
- **Pipeline** de validation automatique
- Pas de push direct



Préfixes normalisés

Conventions de nommage pour une meilleure lisibilité

+ feat/

🔧 fix/

⚙️ chore/

📄 docs/

Bien écrire ses commits



1 commit = 1 idée testable

Chaque commit doit représenter une **modification logique** et testable



Format conseillé

Structure standardisée pour une meilleure lisibilité

type(portée): verbe court



Exemples

feat(api): ajouter /health

fix(ui): corriger overflow mobile



À proscrire

× update

× final

× wip

```
→ commit-msg-linter-test git:(master) ✗ gcam "update README"
```

```
***** Invalid Git Commit Message *****
```

```
commit message: update README
```

```
correct format: <type>[scope]: <subject>
```

```
example: docs: update README to add developer tips
```

```
type:
```

```
feat A new feature.
```

```
fix A bug fix.
```

```
docs Documentation only changes.
```

```
style Changes that do not affect the meaning of the code (white-space, formatting,
```

```
refactor A code change that neither fixes a bug nor adds a feature.
```

```
test Adding missing tests or correcting existing ones.
```

```
chore Changes to the build process or auxiliary tools and libraries such as docume
```

```
perf A code change that improves performance.
```

```
ci Changes to your CI configuration files and scripts.
```

```
build Changes that affect the build system or external dependencies (example scope
```

```
temp Temporary commit that won't be included in your CHANGELOG.
```

```
scope:
```

```
Optional, can be anything specifying the scope of the commit change.
```

```
For example $location, $browser, $compile, $rootScope, ngHref, ngClick, ngView, etc.
```

```
In App Development, scope can be a page, a module or a component.
```

```
subject:
```

```
Brief summary of the change in present tense. Not capitalized. No period at the end.
```

**GIT COMMIT
MESSAGE
GUIDELINES**

Lire l'historique efficacement



Vue d'ensemble

Visualiser l'**historique complet** avec une représentation graphique

```
$ git log --oneline --graph --decorate --all
```



Détail d'un commit

Examiner le **contenu exact** d'un commit spécifique

```
$ git show <sha>
```



Traçabilité par ligne

Identifier **qui a modifié** chaque ligne d'un fichier

```
$ git blame <fichier>
```

⚠ À utiliser avec tact : l'objectif est de comprendre, pas de blâmer

Synchroniser sans bruit



Récupérer les changements

Télécharge les informations du **dépôt distant** sans modifier votre branche locale

```
$ git fetch origin
```

💡 Pratique pour vérifier les nouveautés avant de fusionner



Avancer proprement

Fusionne uniquement si possible en **fast-forward** (sans créer de commit de fusion)

```
$ git merge --ff-only origin/main
```

💡 Garde un historique linéaire et propre



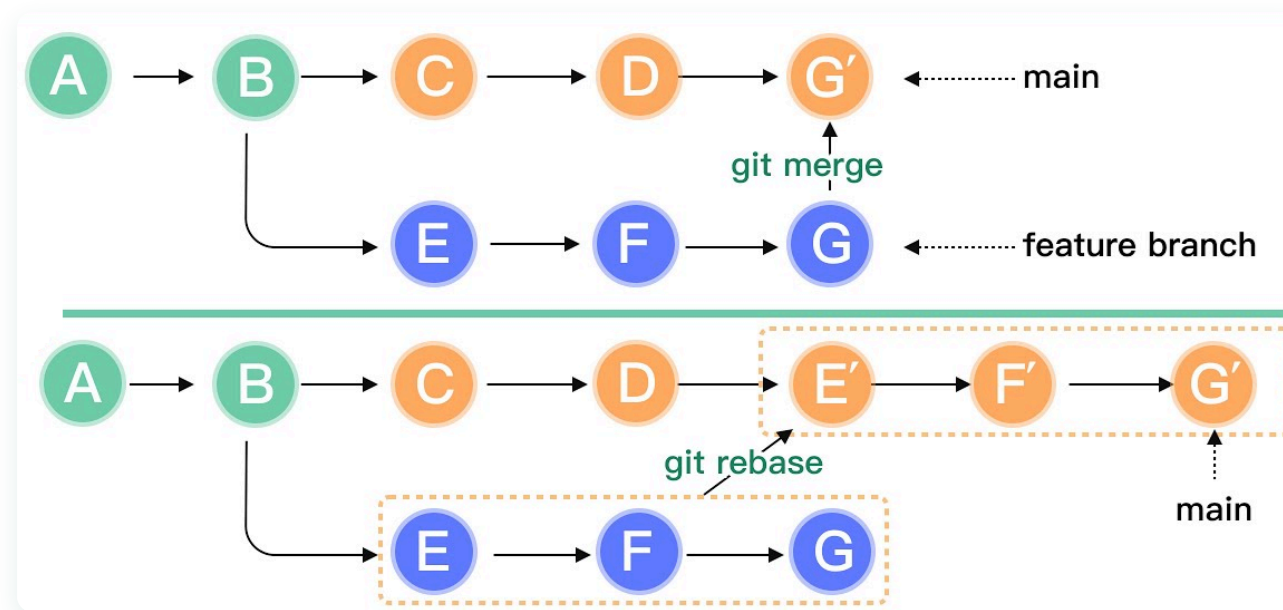
Lisser l'historique local

Rebase votre branche sur la dernière version de **main** pour un historique linéaire

```
$ git rebase origin/main
```

⚠️ N'utilisez que sur des branches privées, jamais sur main

Merge vs Rebase (schéma)



Rebase

Linéarise une branche perso en ré-appliquant les commits sur la branche cible

👍 Avantages

- Historique linéaire
- Plus facile à suivre
- Pas de commits de fusion

👎 Inconvénients

- Réécrit l'historique
- Conflits plus complexes
- Risque de perte de contexte



Merge

Fusionne des branches divergentes en créant un commit de fusion

👍 Avantages

- Préserve l'historique
- Conflits plus simples
- Contexte préservé

👎 Inconvénients

- Historique non linéaire
- Commits de fusion
- Plus difficile à suivre



Règle d'or

Rebase pour les branches privées (votre travail seul) • **Merge** pour les branches publiques (partagées avec l'équipe)

Gérer les conflits (procédure)

1



Identifier

Repérer les fichiers en conflit avec **git status**

```
$ git status
```

```
<<<<<< HEAD
Votre code
=====
Code distant
>>>>>> branch-name
```

2



Résoudre

Éditer les fichiers pour **fusionner** les changements

```
$ code fichier_conflit.js
```

💡 Supprimez les marqueurs <<<<<<, =====, >>>>>>

3



Continuer

Valider les changements et **finaliser** la fusion

```
$ git add .
$ git merge --continue
$ git rebase --continue
```

4



Annuler

Si nécessaire, **abandonner** la fusion en cours

```
$ git merge --abort
$ git rebase --abort
```

⚠️ Utilisez seulement si vous êtes sûr de vouloir annuler

Annuler / corriger sans casser



Corriger message



Modifier le **dernier message** de commit

```
$ git commit --amend -m "Nouveau message"
```



Retirer du stage



Enlever un fichier de la **zone de staging**

```
$ git restore --staged <fichier>
```



Annuler local



Restaurer un fichier à son **dernier commit**

```
$ git restore <fichier>
```



Historique public



Créer un commit **inverse** pour annuler

```
$ git revert <sha>
```



Danger



Réinitialiser brutalement l'historique

```
$ git reset --hard <sha>
```

.gitignore & .gitattributes



.gitignore

Fichier pour **ignorer** les artefacts et fichiers temporaires

```
# Dépendances
node_modules/
.venv/
vendor/

# Build
build/
dist/
*.class

# Environnement
.env
config.local.js
```

📁 node_modules/

📁 .venv/

📁 build/

📄 .env



.gitattributes

Fichier pour **normaliser** les comportements Git entre environnements

```
# Normaliser les fins de ligne
* text=auto

# Fichiers binaires
*.png binary
*.jpg binary

# Fichiers spécifiques
*.md text eol=lf
*.bat text eol=crlf
```



Utilisez Git LFS pour les gros fichiers binaires (images, vidéos...)

Tags & releases (pont vers Build)



Points de référence pour le déploiement

Les tags marquent des **versions spécifiques** du code, servant de pont entre le développement et le processus de build



Créer un tag

Marquer un commit spécifique comme une **version**

```
$ git tag -a v1.0.0 -m "Release initiale"
```



Pousser un tag

Partager le tag avec le **dépôt distant**

```
$ git push origin v1.0.0
```



Semantic Versioning (SemVer)

Standard de numérotation pour une **lisibilité** accrue et une **intégration CI** simplifiée

1 v1.0.0

+ v1.1.0

🔧 v1.1.1

⚙️ v2.0.0

PR/MR de qualité (checklist)



Pull/Merge Requests efficaces

Une PR/MR de qualité facilite la **révision**, accélère la **fusion** et améliore la **traçabilité**

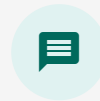


Petite diff

Modifications **ciblées** et **compréhensibles**



Idéalement < 300 lignes pour une révision efficace



Messages clairs

Titre **explicite** et description **détaillée**



Utilisez le format Conventional Commits



Tests validés

Tests et **lint** OK localement



Exécutez la suite de tests avant de créer la PR



Contexte complet

Description + **issue liée**



Référez les tickets avec des mots-clés (fixes #123)

Règles d'équipe (politiques légères)



Protection de main

Branche principale **protégée** pour garantir la stabilité



Pas de **push direct** sur main



Interdiction de **force-push**



Review obligatoire avant fusion



Limites de temps

Maintenir un flux de travail **rapide** et **itératif**



Branches **< 3 jours** de vie



PR/MR **< 300 lignes** de changements



Révision **< 24h** après création



Conventions

Standardisation pour une meilleure **lisibilité**



Noms de branches : **type/sujet**



Messages de commit : **Conventional Commits**



Descriptions de PR/MR **structurées**



Sécurité

Protection des **données sensibles**



Secrets interdits dans le dépôt



Fichiers .env dans **.gitignore**



Utilisation de **vault/CI** pour secrets



Un secret poussé accidentellement doit être révoqué immédiatement

Anti-patterns à éviter



Pratiques à proscrire

Ces **mauvaises habitudes** compliquent la collaboration et créent des problèmes techniques



Mega-commits

Commits "fourre-tout" avec **plusieurs fonctionnalités** mélangées

💡 1 commit = 1 idée testable



Branches longues

Branches qui **dérivent** et accumulent des divergences importantes

💡 Limiter à 2-3 jours de vie



Rebase public

Rebase sur une branche **déjà partagée** avec l'équipe

💡 Rebase privé, merge public



Conflits au hasard

Résolution **aléatoire** des conflits sans comprendre les changements

💡 Analysez chaque changement avant de résoudre



Secrets exposés

Pousser des **clés/API/secrets** directement dans le dépôt

💡 Utilisez vault/CI pour les secrets

Récap & suite du pilier



8 réflexes Git essentiels



pull ff-only



branches courtes



commits
atomiques



messages clairs



rebase local



PR concise



conflits = routine



tags pour releases



Prochaine séance (Partie 3)

Développement Continu — Build & Artefacts

Docker propre, image taguée, CI minimale

Pilier de l'Agilité DevOps