

Workflows Git — aperçu et comparatif

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

0.1 Pourquoi des workflows ?	1
0.1.a Vocabulaire clé	1
0.2 Workflow centralisé (style SVN)	2
0.3 Feature branch workflow (branche par fonctionnalité)	2
0.4 Gitflow (Vincent Driessen)	2
0.5 GitHub Flow (simplifié, PR-based) . .	3
0.6 Forking workflow (open-source)	3
0.7 Trunk-based development (TBD) . . .	4
0.8 Comparatif rapide	4
0.9 Intégration CI/CD	4
1 CLI GitHub	4
1.1 Conclusion	4
1.2 Exercice final : Hello world en équipe	4

0.1 Pourquoi des workflows ?

Un workflow Git formalise comment une équipe collabore sur le code : modèle de branches, règles de fusion, revues et cadence des releases. Il rend le processus prévisible, sécurisé et automatisable.

Objectifs essentiels :

- Clarté — conventions de branches et de commits pour savoir qui fait quoi.
- Collaboration — revues obligatoires, petites PRs, responsabilités partagées.
- Qualité — gates CI (tests, lint, scans) avant toute fusion.

- Livraison — stratégie de release et règles de déploiement.

! Important

Choisissez un workflow adapté à votre équipe, formalisez-le, automatisez-le via CI et appliquez-le strictement.

0.1.a Vocabulaire clé

- Workflow : ensemble de pratiques et règles pour utiliser Git (process, roles, conventions).
- Pull request (PR) / Merge request (MR) : demande de fusion d'une branche — inclut review, CI et discussion.
- CI (Intégration continue / Continuous Integration) : pipelines automatiques (tests, lint, build) déclenchés par les pushes/PRs.
- CD (Livraison continue / Continuous Delivery) : automatisation des déploiements après CI réussie.
- Branch protection : règles côté serveur (ex. pas de push direct, exigences de review/CI) pour protéger `main`.
- Conventional Commits : convention pour formater les messages de commit (ex. `feat:`, `fix:`) — utile pour changelogs automatisés.
- Rebase vs Merge : rebase = historique linéaire (réécriture locale), merge = commit de fusion (trace explicite).

0.2 Workflow centralisé (style SVN)

i Principe

une branche principale (main/master) et tout le monde push/pull dessus.

```
Developers -> commit -> push -> origin/main
```

Commandes typiques :

```
# basculer sur la branche principale (moderne)
git switch main
# synchroniser en conservant un historique linéaire local
git pull --rebase
git add .
git commit -m "...
git push origin main
```

- Avantages : simple, facile à comprendre pour débutants.
- Inconvénients : risque élevé de conflits, intégrité limitée, pas de revue obligatoire.

0.3 Feature branch workflow (branche par fonctionnalité)

i Principe

- Chaque fonctionnalité ou bugfix a sa propre branche `feature/x`.
- Fusion via merge ou PR dans main.

```
main ---o-----o-----o
      \         /
feature1 o-----o
```

Commandes :

```
# créer et basculer (moderne)
git switch -c feature/ma-fonctionnalite
# pousser et définir la branche distante par défaut
git push --set-upstream origin feature/ma-fonctionnalite
# pendant le travail: garder votre branche à jour avec `main`
git fetch origin
git rebase origin/main
# après review et tests (ou via PR) :
git switch main
git merge --no-ff feature/ma-fonctionnalite # ou utilisez squash via l'interface PR
git push origin main
```

- Avantages : isole le travail, facilite revue et CI.
- Inconvénients : branches longues peuvent diverger; nécessité de synchroniser (rebase/merge).

0.4 Gitflow (Vincent Driessen)

i Principe

- Branches principales : `main` (release) et `develop` (intégration)
- Branches support : `feature/`, `release/`, `hotfix/`

```
main ---o-----o
      \         /
develop ---o---o---o
        /         \
feature1          release
```

Commandes (exemple simplifié) :

```
# feature
git checkout develop
```

```
git checkout -b feature/x
# push feature
git push -u origin feature/x
# finish feature
git checkout develop
git merge --no-ff feature/x
# release
git checkout -b release/1.2 develop
git checkout main
git merge --no-ff release/1.2
git tag -a v1.2
```

- Avantages : clair pour projets avec versions/stabilisation. Conserve historique de releases.
- Inconvénients : plus complexe, lourdeur pour déploiements rapides.
- Vincent Driessen Gitflow — <http://nvie.com/posts/a-successful-git-branching-model/>
- <https://www.atlassian.com/fr/git/tutorials/comparing-workflows/gitflow-workflow>

0.5 GitHub Flow (simplifié, PR-based)

i Principe

- Branche courte depuis main, ouvrir Pull Request, revue, merge, déploiement continu.
- Pas de develop ni branche de release.

```
main ---o---o---o
      \  /
      PR-merge
```

- Bon pour équipes produisant déploiements fréquents (CI/CD). Favorise petites PRs.
- GitHub Flow — <https://guides.github.com/introduction/flow/>

0.6 Forking workflow (open-source)

i Principe

- Chaque contributeur fork le repo principal, travaille dans son fork, ouvre Pull Request vers le repo parent.

```
Upstream (org/repo)
  ^
  | Pull Request
Fork (user/repo) <- clone
```

```
# clone du fork
git clone git@github.com:utilisateur/repo.git
git remote add upstream git@github.com:org/repo.git
# synchroniser (fetch + rebase upstream pour éviter les merges locaux)
git fetch upstream
git switch main
git rebase upstream/main
# pousser la branche mise à jour sur votre fork si besoin
git push origin main
```

- Avantages : sécurité (pas d'accès direct), isolation des contributeurs, récupération facile des contributions et des mises à jour de l'origine.
- Inconvénients : overhead pour synchroniser forks.

0.7 Trunk-based development (TBD)

i Principe

- Tout le monde intègre fréquemment (plusieurs fois par jour) sur `main` (ou `trunk`), via petites branches très courtes.
- **Pre-integration via CI : tests automatisés avant merge.**
- Favorise feature flags, CI rapide, releases continues.

- Avantages : réduit divergence, facilite CI/CD.
- Inconvénients : **nécessite tests TRES solides, discipline et feature flags.**
- Trunk-based development — <https://trunkbaseddevelopment.com/>

0.8 Comparatif rapide

- Centralisé : très simple, pas recommandé pour équipes collaboratives.
- Feature branch : bon compromis pour la plupart des équipes.
- GitHub Flow : léger, idéal pour déploiements continus.
- Gitflow : adapté aux équipes avec releases planifiées.
- Forking : idéal pour contributions open-source.
- Trunk-based : requis pour livraison continue à grande échelle.

0.9 Intégration CI/CD

- CI: Intégration Continue
 - Chaque push sur une branche active déclenche pipelines (tests, lint, builds).
 - PRs doivent passer la pipeline avant merge.
 - Protect branch rules : exiger revue, tests verts, pas de push direct (par ex. sur `main`).
 - **Test early, fail fast, fix quickly.**
- CD : Livraison Continue

- Déploiement automatisé en production après CI réussie.

1 CLI GitHub

- GitHub CLI (`gh`) facilite La gestion des repositories, PRs, issues, releases depuis terminal, ...

```
# cloner un repo GitHub
gh repo clone utilisateur/repo
# Créer un nouveau repo GitHub
gh repo create mon-nouveau-repo --private
# créer une PR via GitHub CLI
gh pr create --fill --base main --head feature/x
# récupérer et basculer sur la branche d'une PR
gh pr checkout 123
```

1.1 Conclusion

- Tout les projets (même seul) bénéficient d'un workflow clair.
 - Choisissez un workflow adapté à votre équipe et formalisez-le dans le README du projet.
 - Configurez règles de branche et CI pour l'automatisation.
 - Pour commencer, protégez `main` contre les pushes directs et exigez des revues de code via PRs.
 - Automatisez les tests (`./mvnw package` ou `./mvnw verify`) via CI pour chaque PR et à chaque merge sur `main`.

1.2 Exercice final : Hello world en équipe

- Vous allez recevoir un lien vers un dépôt GitHub.
- Clonez-le localement.
- Créez une branche `feature/hello-<votre-login>`.

- Ajoutez une classe Java qui implante l'interface `SayHello` fournie. Votre classe doit retourner «Hello from !» et être dans un package `fr.univtlh.<votre-login>.hellogit`.
- **Compilez et testez localement.**
- Committez vos changements avec un message clair en respectant «Conventional Commits».
 - Le message doit commencer par `feat: add Hello from <votre-login>`
- Poussez votre branche sur le dépôt distant.
- Ouvrez une Pull Request vers `main`.
- Demandez à un autre étudiant de revoir votre code et de l'approuver.
- Une fois approuvée, fusionnez la PR.