

Git - Bonnes pratiques et conventions

Comprendre les conventions de dénomination des branches Git

Les conventions de dénomination des branches dans Git fournissent un moyen systématique d'organiser et de référencer les branches au sein d'un référentiel. Ces conventions ne sont pas appliquées par Git lui-même, mais peuvent être mises en œuvre via des politiques d'équipe ou des scripts automatisés. Une dénomination efficace des branches est essentielle pour plus de clarté, en particulier dans les projets avec plusieurs contributeurs.

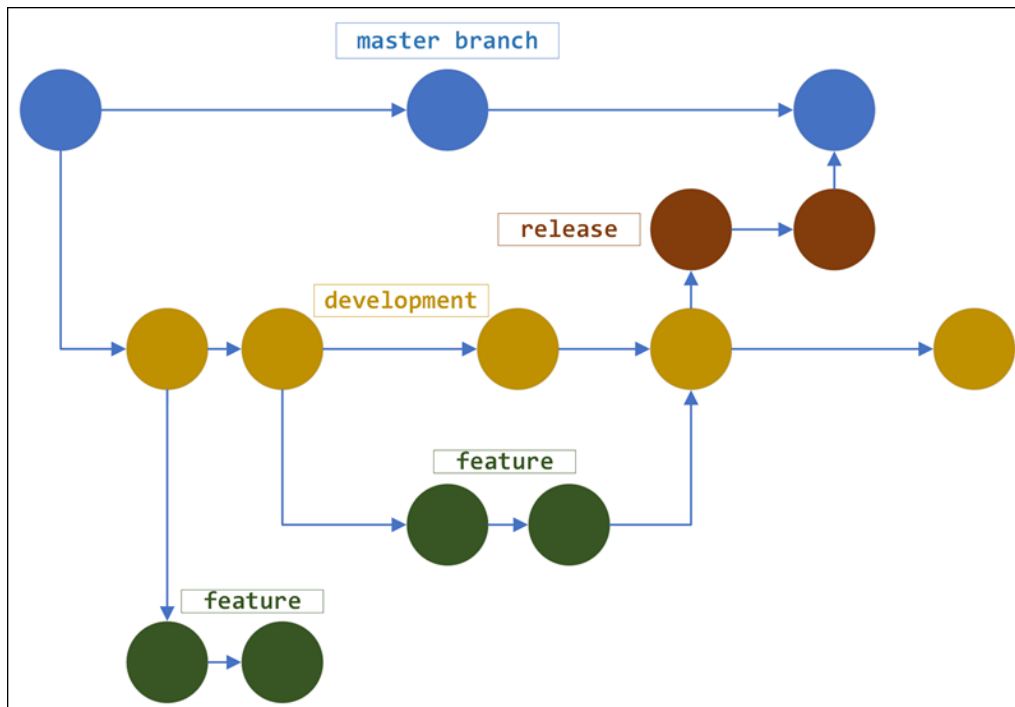
Convention de dénomination des branches

Types de branches Git

Les branches dans Git peuvent être classées en deux types en fonction de leur longévité et du type de changement qu'elles introduisent :

1. **Branches régulières**
2. **Branches temporaires**

Cette catégorisation permet aux équipes de mieux gérer leurs flux de travail Git. Une dénomination appropriée garantit que les branches à longue durée de vie restent stables et que les branches temporaires sont utilisées pour des tâches spécifiques à court terme sans encombrer l'historique du référentiel.



Les sections ci-dessous expliquent les deux types en détail.

Branches Git régulières

Les branches régulières dans Git sont des branches à longue durée de vie qui constituent un moyen stable et structuré d'organiser le travail en cours. Ces branches sont disponibles en permanence dans le dépôt et leur nommage est simple.

Les branches Git régulières les plus utilisées sont :

- **Branche principale `main`**: c'est la branche de production par défaut dans un dépôt Git qui doit être stable en permanence. Les développeurs ne peuvent fusionner les modifications dans la branche principale qu'après révision et test du code. Tous les collaborateurs d'un projet doivent maintenir la branche principale stable et à jour.
- **Branche de développement `develop`**: c'est la branche de développement principale qui sert de plaque tournante centrale aux développeurs pour intégrer de nouvelles fonctionnalités, des corrections de bugs et d'autres modifications. Son objectif principal est d'être le lieu où apporter des modifications pour empêcher les développeurs de les implémenter directement dans la branche principale. Les développeurs testent, examinent et fusionnent les modifications de la branche `develop` dans la branche principale `main`.
- **Branche QA / de test `test`**: c'est la branche qui contient tout le code prêt pour les tests d'assurance qualité et d'automatisation. Les tests d'assurance qualité sont nécessaires avant d'implémenter tout changement dans l'environnement de production pour maintenir une base de code stable.

Branches Git temporaires

Les branches temporaires sont éphémères et jetables. Ces branches servent à des fins spécifiques à court terme et sont souvent supprimées par la suite.

Voici les branches temporaires courantes de Git recommandées :

- **Branches de fonctionnalités** : ces branches sont utilisées pour développer de nouvelles fonctionnalités. Utilisez le préfixe `feature/`. Par exemple, `feature/login-system`.
- **Branches de correction de bogues** : ces branches sont utilisées pour corriger les bogues dans le code. Utilisez le préfixe `bugfix/`. Par exemple, `bugfix/header-styling`.
- **Branches de correctifs** : ces branches sont créées directement à partir de la branche de production `*main` pour corriger les bogues critiques dans l'environnement de production. Utilisez le préfixe `hotfix/`. Par exemple, `hotfix/critical-security-issue`.
- **Branches de publication** : ces branches sont utilisées pour préparer une nouvelle version de production. Utilisez le préfixe `release/`. Par exemple, `release/v1.0.1`.
- **Branches de documentation** : ces branches sont utilisées pour écrire, mettre à jour ou corriger la documentation. Utilisez le préfixe `docs/`. Par exemple, `docs/api-endpoints`.

Règles de nommage de branches Git

Git impose quelques restrictions de base sur les noms de branches :

- **Caractères** : Les noms de branches peuvent inclure des lettres, des chiffres, des tirets (-), des traits de soulignement (_) et des points (.), *mais ils ne peuvent pas commencer par un point ni se terminer par une barre oblique (/)*.
- **Sensibilité à la casse** : Git est sensible à la casse, donc **Feature/** et **feature/** sont considérés comme des branches différentes. En pratique, il est plus conventionnel d'utiliser uniquement des minuscules.
- **Noms réservés** : les noms comme **HEAD**, **FETCH_HEAD**, **ORIG_HEAD**, et autres sont réservés par Git et ne peuvent pas être utilisés comme noms de branches.
- **Longueur** : Bien qu'il n'y ait pas de limite stricte sur la longueur des noms de branches, il est pratique de les garder concis et courtes pour les rendre plus faciles à gérer.

Exemples

```
# Creating a new feature branch
git checkout -b feature/add-user-authentication

# Switching to a bugfix branch
git checkout -b bugfix/login-issue

# Creating new release branch
git checkout -b release/v2.0.0

# Preparing a hotfix
git checkout -b hotfix/reset-password-fix
```

Dans chaque commande ci-dessus, **git checkout -b** est utilisé pour créer et basculer vers une nouvelle branche en une seule commande *git*, avec le nom de la branche suivant immédiatement la commande indiquant son objectif et sa structure.

Bonnes pratiques pour l'utilisation des balises Git (tag)

- Utiliser le [contrôle de version sémantique \(SemVer\)](#) - voir ci-dessous.
- Préfixez le numéro de version avec la lettre **v**.

Exemples

```
git tag -a v1.0.0
git tag -a v2.1.7
```

Résumé de la spécification de la gestion sémantique de version (SemVer)

Étant donné un numéro de version **MAJEUR.MINEUR.CORRECTIF** :

MAJEUR

Une modification **MAJEUR** est une modification qui modifie la manière dont le logiciel fonctionne d'une manière qui est incompatible avec les versions précédentes. Cela peut inclure des changements dans les interfaces API, des suppressions de fonctionnalités, ou d'autres modifications qui nécessitent une action de la part de l'utilisateur pour continuer à utiliser le logiciel.

MINEUR

Une modification **MINEUR** ajoute des fonctionnalités ou des améliorations qui sont compatibles avec les versions précédentes. Par exemple, l'ajout d'une nouvelle méthode à une API sans supprimer ou modifier les méthodes existantes serait une modification mineure.

CORRECTIF

Un **CORRECTIF** ou patch est une modification qui corrige des bugs ou des problèmes de sécurité, et qui est conçue pour être totalement compatible avec les versions précédentes. Cela signifie que vous pouvez appliquer le patch sans affecter la fonctionnalité existante.

Un numéro de version standard doit prendre la forme **X.Y.Z** où **X**, **Y** et **Z** sont des entiers non négatifs et ne doivent pas PAS être préfixés par des zéros (ex: 01).

- **X** représente l'identifiant de version majeure,
- **Y** représente l'identifiant de version mineure,
- **Z** l'identifiant de version de correction / patch.

Chaque élément doit s'incrémenter numériquement. Exemple : **1.9.0** -> **1.10.0** -> **1.11.0**.

Comment dois-je gérer les révisions dans la phase initiale de développement 0.y.z ?

La chose la plus simple à faire est de commencer vos développements avec une version initiale à 0.1.0 puis d'incrémenter l'identifiant de version mineure pour chaque nouvelle publication.

Comment savoir quand publier la version 1.0.0 ?

Si votre logiciel est utilisé en environnement de production ou que vous avez une application stable de laquelle des utilisateurs ont commencé à dépendre, vous devriez probablement déjà être en version 1.0.0. Et si vous vous faites déjà du souci pour la rétro-compatibilité, vous devriez également avoir dépassé la 1.0.0.

Exemple GitHub Action: `uses: actions/setup-node@v3`

Cette ligne signifie que vous utilisez l'action `setup-node` de GitHub Actions et que vous ciblez la **version majeure 3**.

Cela signifie que votre workflow utilisera la dernière version mineure et le dernier patch qui correspondent à cette version majeure.

Dans un contexte pratique, si les versions disponibles de `setup-node` sont :

- 3.0.0
- 3.1.0
- 3.1.1
- 3.2.0
- 4.0.0

Utiliser `@v3` prendrait automatiquement la dernière version compatible avec la version majeure 3, qui serait 3.2.0. Notez que la version 4.0.0 ne serait pas prise car elle appartient à une nouvelle version majeure.

Ceci vous permet d'accepter automatiquement les mises à jour mineures et les patches, tout en évitant les changements majeurs qui pourraient casser votre configuration actuelle. Gare bien cela en tête pour la suite de la formation !