

Git Flow : Principe et Intérêt pour la CI/CD

1. Introduction

Le **Git Flow** est un modèle de gestion de branches Git conçu pour structurer le développement logiciel, surtout dans les projets avec des cycles de release bien définis. Il permet de :

- Séparer clairement le code stable (production) du code en développement.
- Faciliter la collaboration entre plusieurs développeurs.
- Automatiser les tests et déploiements (CI/CD).

2. Branches principales

Branche	Rôle	Exemple d'utilisation
main	Contient le code de production, toujours stable et prêt à être déployé.	Version disponible sur les stores (App Store, Google Play, etc.).
dev	Intègre les nouvelles fonctionnalités et corrections avant la release.	Branche où tout le développement actif a lieu avant de fusionner vers main .

3. Branches de support (temporaires)

a. Feature branches

- **Origine** : **dev**
- **Préfixe** : **feature/*** (ex: **feature/login-page**)
- **Utilité** : Développer une nouvelle fonctionnalité de manière isolée.
- **Fusion** : Vers **dev** après validation.

Exemple :

```
git checkout -b feature/login-page dev
# Développement...
git checkout dev
git merge --no-ff feature/login-page
git branch -d feature/login-page
```

Bash

b. Release branches

- **Origine** : **dev**
- **Préfixe** : **release/*** (ex: **release/1.2.0**)
- **Utilité** : Préparer une nouvelle version (corrections de bugs mineurs, documentation, etc.).
- **Fusion** : Vers **main** (avec un tag de version) et **dev**.

Exemple :

```
git checkout -b release/1.2.0 dev
# Corrections de bugs, tests finaux...
git checkout main
git merge --no-ff release/1.2.0
git tag -a v1.2.0
```

Bash

```
git checkout dev
git merge --no-ff release/1.2.0
```

c. Hotfix branches

- **Origine** : `main`
- **Préfixe** : `hotfix/*` (ex: `hotfix/critical-bug`)
- **Utilité** : Corriger un bug critique en production sans attendre le cycle de release.
- **Fusion** : Vers `main` (avec un tag) et `dev`.

Exemple :

```
git checkout -b hotfix/critical-bug main
# Correction du bug...
git checkout main
git merge --no-ff hotfix/critical-bug
git tag -a v1.2.1
git checkout dev
git merge --no-ff hotfix/critical-bug
```

Bash

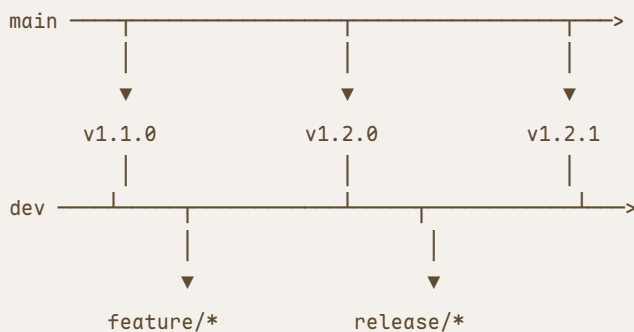
4. Intérêt pour la CI/CD

- **Automatisation des tests** : Chaque fusion vers `dev` ou `main` peut déclencher des pipelines CI (tests unitaires, intégration, etc.).
- **Déploiement continu** : La branche `main` déclenche un build et une publication automatique sur Expo et les stores.
- **Isolation des fonctionnalités** : Les `feature branches` permettent de tester une fonctionnalité en isolation avant intégration.
- **Gestion des versions** : Les `release branches` et les tags facilitent le suivi des versions et les rollbacks si nécessaire.

5. Workflow complet

1. Un développeur crée une `feature/login-page` depuis `dev`.
2. Après validation, la feature est mergée dans `dev`.
3. Quand `dev` est stable, une `release/1.2.0` est créée pour les tests finaux.
4. La release est mergée dans `main` (avec un tag) et `dev`, puis déployée automatiquement via EAS.
5. Si un bug critique est détecté en production, un `hotfix` est créé depuis `main`, corrigé, puis mergé dans `main` et `dev`.

6. Schéma visuel simplifié



7. Bonnes pratiques pour React Native + Expo

- **Utiliser EAS Build** pour générer des builds à chaque tag sur `main`.
- **Configurer des canaux de release** dans Expo pour tester les `feature branches` :

```
npx expo publish --release-channel dev
```

Bash

- **Automatiser la génération des notes de release** à partir des commits depuis le dernier tag.
- **Utiliser des variables d'environnement** pour gérer les clés API et les configurations spécifiques à chaque environnement (dev/prod).

