ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

# Travailler avec Git

## Le Git flow centralisé

## 1. Introduction

Un workflow définit un plan d'organisation du travail de collaboration de plusieurs développeurs pour la production de logiciels. La version de workflow présentée ici est le *Git-flow centralisé*. La collaboration se fait par l'intermédiaire d'un dépôt centralisé que partagent tous les développeurs et des dépôts locaux de travail propres à chaque développeur.

Dans la description qui suit, le dépôt central du projet est supposé localisé sur gitlab.ecole.ensicaen.fr. Les commandes décrites sont les commandes de base de Git. Il existe des commandes spécifiques du *Git-flow* qui ne sont pas utilisées ici. Les commandes sont aussi directement exécutables dans un Environnement de Développement Intégré digne de ce nom (p.ex. Intellij, Eclipse, Visual Studio Code).

## 2. Organisation du dépôt central

Le développement d'un projet s'organise typiquement autour de cinq catégories de branche, dont la Figure 1 donne un exemple.

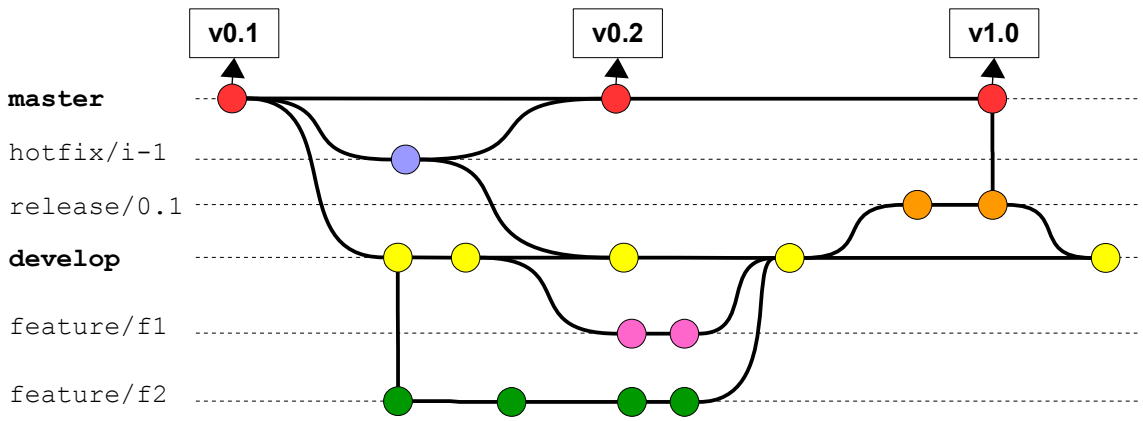



**Figure 1: Organisation des branches sur le dépôt central.**

## 2.1 Une branche permanente de production : *master*

La branche `master` contient uniquement les versions stables du logiciel. Ce sont les différentes versions publiées. Elles sont taguées en conséquence (*vX.Y*). Seul le responsable du dépôt peut modifier cette branche. Elle doit rester propre avec un historique lisible retraçant les étapes clés du développement.

## 2.2 Une branche permanente d'intégration : *develop*

La branche `develop` sert d'intégration des différentes productions des développeurs pour construire la prochaine version dans `master`. Le responsable du dépôt y fusionne les branches de développement et de correction de bugs des autres développeurs. Là encore, la branche doit rester saine en évitant la pollution des historiques des développements des fonctionnalités.

## 2.3 Des branches temporaires de développement : *feature/ff*

Chaque développeur crée une branche spécifique pour développer la fonctionnalité qui lui est attribuée. La branche est nommée `feature/ff` où `ff` est un nom explicite en référence à la fonctionnalité, par exemple `feature/query-database`. La branche dure le temps du développement de la fonctionnalité. Quand la fonctionnalité est complète, elle est poussée sur le dépôt central et le responsable du dépôt la fusionne à la branche `develop` courante, puis la branche est détruite par le développeur.

Cette branche réside essentiellement sur le dépôt local du développeur jusqu'à la fin du développement où elle sera alors poussée sur le dépôt commun. Néanmoins, elle peut aussi être poussée régulièrement sur le dépôt central pour assurer des sauvegardes intermédiaires et échanger avec les autres développeurs.

## 2.4 Une branche temporaire de mise en production : *release/x.y*

Quand la branche `develop` doit passer en production, une branche `release/x.y` est détachée de la branche `develop`. La création de cette branche commence le cycle de production de la prochaine version *vX.Y*. Aucune nouvelle fonctionnalité ne peut y être ajoutée. Seule la correction de bugs, la génération

de documentation et d'autres tâches axées sur la version peuvent venir modifier cette branche. Une fois que la version est prête à la production, elle est fusionnée à `master` et étiquetée avec un numéro de version. Elle est aussi fusionnée à la branche `develop` qui peut avoir progressé depuis que la publication a été lancée. Cette branche est ensuite supprimée.

Avant de poursuivre leur développement, les développeurs rebasent leur travail en cours avec la version de `develop`.

### 2.5 Des branches temporaires de dépannage de bug en production : *hotfix/issue-x*

La correction d'un bug découvert dans une version en production est confiée à un développeur. La correction se fait dans une branche spécifique construite à partir de la version de `master`. La branche est généralement nommée `hotfix/issue-x` où `x` est le numéro du bug. Un hotfix est un patch qui va s'appliquer directement à la branche de production `master` et qui sera ensuite également appliqué à la branche d'intégration `develop`.

Lorsque le correctif est appliqué, le responsable du dépôt fusionne la branche `hotfix/issue-x` à la branche `master` avec le numéro de version qui convient, ainsi qu'à la branche `develop` (et la branche `release` en cours le cas échéant) pour mettre à jour les modifications apportées. Encore une fois, il est préférable que tous les développeurs rebasent leur travail en cours avec la version de `develop` avant de continuer leur développement.

## 3. Le workflow

Examinons comment une petite équipe formée de trois développeurs, Estelle, Sébastien et Loïc collaborent en utilisant le workflow Git via un dépôt centralisé sur Gitlab. Estelle est la responsable du dépôt centralisé.

### 3.1 Initialisation du dépôt

#### 3.1.1 Estelle initialise le dépôt central sur Gitlab

Estelle crée le dépôt central sur Gitlab. Elle ajoute Loïc et Sébastien comme

développeurs. Elle protège la branche `master` contre la modification par les autres développeurs. Estelle sera la seule à gérer cette branche pour garantir l'unité du projet. Estelle réalise simplement ces opérations par l'intermédiaire de l'interface Web de Gitlab.

### 3.1.2 Estelle crée la branche de développement sur Gitlab

Toujours par l'intermédiaire de l'interface de Gitlab, Estelle crée la branche `develop` par dérivation de la branche `master` qu'elle protège aussi contre toute modification par d'autres développeurs.

### 3.1.3 Tous les développeurs clonent le dépôt central

Les trois développeurs créent leur dépôt local par copie du dépôt central. Toutes les opérations suivantes peuvent se réaliser par lignes de commande ou directement à partir de l'interface d'un IDE.

```
git clone https://estelle@gitlab.ecole.ensicaen.fr/estelle/depot.git
```

## 3.2 Développement des fonctionnalités

### 3.2.1 Estelle, Loïc et Sébastien commencent leur travail de développement

Chaque développeur crée une branche spécifique pour développer la fonctionnalité qui lui est attribuée. La branche est d'abord une copie de `develop`. Loïc crée la branche `feature/query-database`.

```
git checkout develop
git checkout -b feature/query-database
```

### 3.2.2 Loïc développe sa fonctionnalité

Sur la branche `feature/query-database`, Loïc édite, crée, modifie et supprime des fichiers du projet avec son IDE pour ajouter sa fonctionnalité. En suivant le principe de petits pas, il fait autant de *commits* que nécessaire pour garder trace de ses différentes versions et pouvoir revenir en arrière en cas de nécessité.

***Remarque***

En général, Loïc commit directement tous les fichiers modifiés dans le dossier de travail.

```
git add -A
git commit -m "Descriptif du commit"
```

Mais, il arrive que lorsque Loïc travaille sur une fonctionnalité, il corrige une petite erreur qui n'a rien à voir avec la fonctionnalité en cours, par exemple un bug ou une faute d'orthographe dans un commentaire. Dans ce cas, il pousse uniquement le fichier modifié dans la zone d'index (*staging area*) et il fait un commit distinct.

```
git add DataAccessObject.java
git commit -m "Correction du bug d'accès aux données"
```

Ainsi, la correction est prise en compte mais n'est pas associée à des évolutions de la fonctionnalité si Loïc décide de supprimer ces dernières par la suite.

### 3.2.3 Loïc finit sa journée de travail

Avant de partir, Loïc pousse sa branche sur le dépôt central.

```
git add -A
git commit -m "Descriptif du commit"
git push origin feature/query-database
```

Cela sert de sauvegarde, et si Loïc collabore avec d'autres développeurs, cela sert aussi à leur donner accès à ses commits.

## 3.3 Publication d'une fonctionnalité

### 3.3.1 Loïc termine sa fonctionnalité

Avant d'envoyer son travail pour intégration au projet, Loïc doit synchroniser sa branche avec la branche `develop` courante du dépôt central qui a pu évoluer depuis qu'il en a dérivé. Cela nécessite de résoudre tous les problèmes de conflit entre les deux versions. Il doit aussi s'assurer que ses tests passent avec la version synchronisée. L'objectif est de faciliter le travail de fusion ultérieur d'Estelle afin qu'elle n'est pas à lire le code en profondeur pour résoudre les conflits ou les bugs résultants.

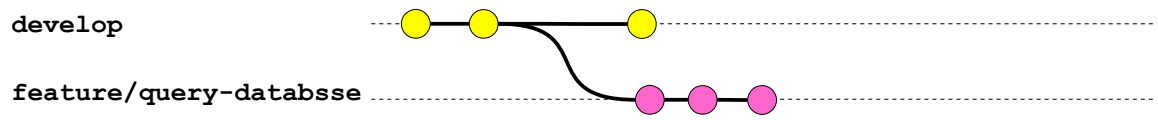
### 3.3.2 Loïc synchronise sa branche avec *develop*

La synchronisation se fait en changeant la base de la branche de fonctionnalité de Loïc avec la tête courante de la branche `develop` comme si la branche de fonctionnalité venait juste de dériver de la branche `develop`.

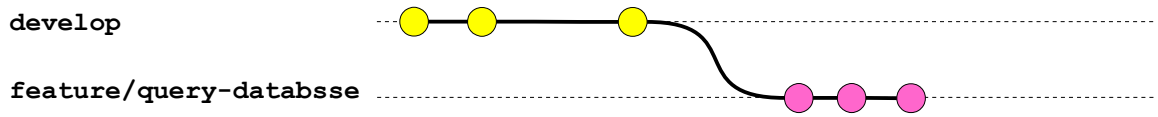
Pour cela, Loïc utilise la commande `rebase` de sa branche sur la branche `develop` :

```
git checkout feature/query-database
git rebase -i develop
```

La commande `rebase` a pour effet de modifier le départ de la branche :

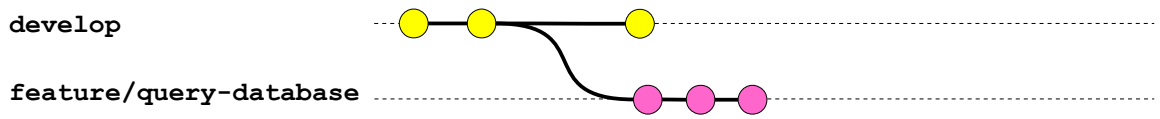



Après le `rebase` :

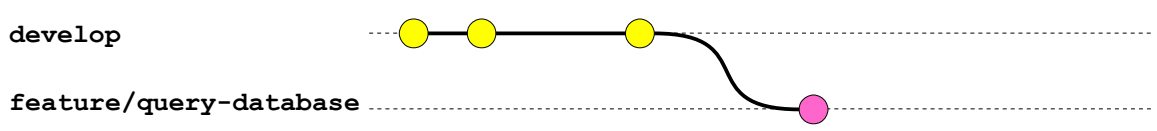



L'option `-i` de `git rebase` permet de nettoyer l'historique interactivement en fusionnant les commits intermédiaires et en réorganisant les commits dans un ordre plus explicite. Quand Loïc développe, il fait beaucoup de commits intermédiaires selon le principe du petit pas. Il se retrouve donc avec un historique comportant beaucoup de commits intermédiaires non utiles avec des messages de commit du genre « *refactoring de la BD* », « *encore le refactoring de la BD* », « *toujours le refactoring* de la BD ».

Loïc fait son rebase interactif pour regrouper les trois commits ci-dessous en un seul « *Refactoring de la BD* » :




La modification totale, `rebase` plus regroupement des commits, donne :



### 3.3.3 Loïc envoie son travail à Estelle

Une fois la résolution des conflits effectuée et les tests passés, Loïc pousse sa branche sur le dépôt central :

```
git add -A
git commit -m "Descriptif du commit"
git push origin feature/query-database
```

Puis il remplit un « *merge request* » dans Gitlab pour demander à Estelle de fusionner sa branche dans `develop`.

***Remarque***

- `git add -A` : ajoute tout.
- `git add .` : Ajouter les fichiers nouveaux et modifiés, mais pas les fichiers détruits.
- `git add -u` : ajouter les fichiers modifiés et détruits, mais pas les nouveaux.

### 3.3.4 Estelle reçoit le « merge request » de Loïc

Quand Estelle reçoit la demande d'intégration, elle contrôle la qualité de la contribution, notamment le respect des conventions de codage et de la présence des tests. Elle peut décider de modifier ou de faire modifier la contribution par Loïc avant l'intégration dans le projet officiel.

### 3.3.5 Loïc effectue les changements

Pour effectuer les modifications, Loïc utilise exactement le même processus qu'il a fait pour créer la première itération de sa fonction. Il édite, valide et pousse les mises à jour sur le dépôt central. Toutes ses activités apparaissent dans le « *merge request* » et Estelle en est informée.

### 3.3.6 Loïc termine le travail de révision de sa fonctionnalité

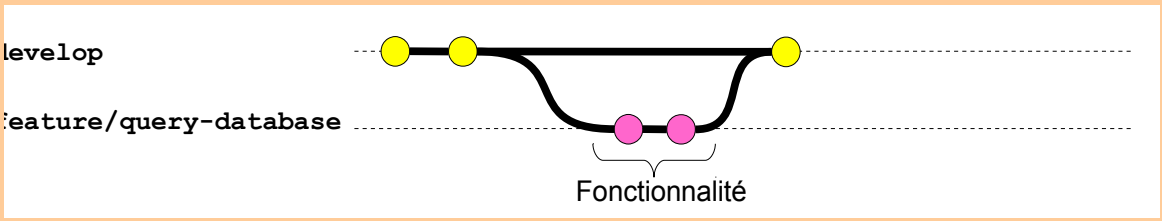
Une fois que la contribution de Loïc est jugée acceptable par Estelle, elle fusionne la contribution sur la branche `develop` en utilisant simplement le bouton "Accepter" de l'interface de Gitlab, ou bien les lignes de commandes :

```
git pull origin develop
git checkout develop
git merge -no-ff feature/query-database
git push
```

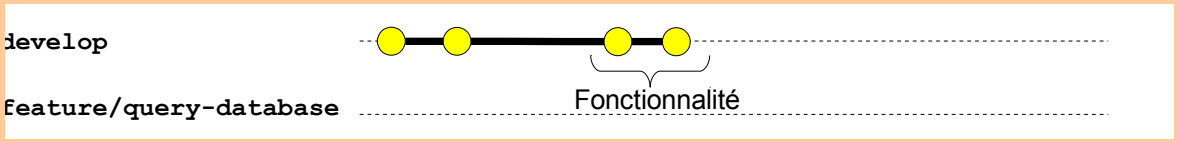
La fusion doit utiliser l'option `--no-ff` de manière à garder trace de la branche `feature/query-database` dans l'historique de la branche `develop`.

***Remarque***

L'option `--no-ff` (*no fast-foward*) force la fusion à toujours créer un nouvel objet commit, même quand le merge peut se faire avec fast-forward. Cela permet de ne pas perdre l'information sur l'existence historique d'une branche de fonctionnalité et regrouper ensemble tous les commits qui sont ajoutés à la branche.




En *fast-forward*, la branche est ajoutée dans `master` comme des commits classiques. Il n'y a plus trace de la branche. Il est impossible de voir à partir de l'historique quels commits font partie de la fonctionnalité ajoutée. Il est aussi difficile de revenir en arrière sans relire précautionneusement tous les messages de commit pour annuler cette fonctionnalité.




### 3.3.7 Estelle ajoute avec succès la fonctionnalité de Loïc à la branche « develop »

Après avoir réalisé la synchronisation avec le dépôt central, Estelle publie la version de `develop`.

Loïc supprime sa branche avant d'en commencer une nouvelle :

```
git branch -d feature/quesry-database
```

Sébastien, comme tous les autres développeurs rebase sa branche courante sur la branche `develop` :

```
git pull origin develop
git checkout feature/seb
git rebase develop
```

Il règle tous les problèmes de conflit pour se retrouver avec une branche qui intègre toutes les fonctionnalités et les tests à jour.

Il pousse ensuite sa branche sur le dépôt central avec l'option `-f` pour forcer

la mise à jour malgré le changement de base :

```
git push -f
```

## 3.4 Passage en production

### 3.4.1 Estelle commence la mise en production d'une version stable du logiciel

Alors que Sébastien travaille toujours sur sa fonctionnalité et que Loïc débute une nouvelle fonctionnalité sur une nouvelle branche, Estelle commence à préparer la nouvelle version officielle du projet. Elle utilise une nouvelle branche dérivée de `develop` pour encapsuler la préparation. La branche est nommée `release/x.y` où `x.y` est le numéro de la version :

```
git checkout develop

git checkout -b release/0.1
```

Cette branche est l'endroit où nettoyer le projet, tout tester, mettre à jour la documentation et stabiliser la version.

Dès qu'Estelle crée cette branche, la version est gelée. Toute fonctionnalité qui n'est pas déjà dans `develop` est reportée au cycle de développement suivant.

### 3.4.2 Estelle publie la version

Une fois que la version est prête, Estelle fusionne sa branche de développement `release/x.y` dans `master` et `develop`, puis supprime la branche. Il est important de fusionner de nouveau dans `develop`, car des mises à jour critiques peuvent avoir été ajoutées à la branche et elles ont besoin d'être accessibles aux nouvelles fonctionnalités dans la branche `develop`.

```
git pull origin master
git checkout master
git merge --no-ff release/v0.1
git tag -a v0.1

git pull origin develop
git checkout develop
git merge -no-ff release/v0.1
git branch -d release/v0.1
```

Là encore, la fusion doit se faire sans `fast-forward`, d'où l'option `-no-ff`.

Tous les développeurs rebasent alors leur branche en cours sur `develop` :

```
git pull origin develop
git rebase develop
```

## 3.5 Dépannage de bug en production

### 3.5.1 Estelle corrige le bug

Un utilisateur final fait remonter un bug dans la version actuelle. Estelle crée une branche de maintenance, par exemple `hotfix/issue-001` à partir de `master`, résout le problème avec autant de commits que nécessaire, puis fusionne le résultat directement dans `master` et `develop` (ou `release` à la place de `develop` si une version est en cours de préparation en même temps). Enfin, la branche est supprimée.

```
git checkout -b hotfix/issue001
git commit -A -m "Fig bug on ..."
```

À la fin du travail de correction, les modifications sont répercutés sur les branches `master` et `develop` (ou `release`) :

```
git pull origin master
git checkout master
git merge --no-ff hotfix/issue001
git tag -a v0.2

git pull origin develop
git checkout develop
git merge --no-ff hotfix/issue001

git branch -d hotfix/issue001
```

Si la branche `develop` a été modifiée, tous les développeurs rebasent leur travail :

```
git pull origin develop
git rebase develop
```