Branches

Le but des branches est de permettre à plusieurs développeurs de travailler, en même temps sur un projet, tout en créant des sauvegardes régulières de leurs sources.

Comment c'était avant git ?

Ou plutôt, comment c'était avant les systèmes de versioning ?

Mise en commun manuelle

Tous les développeurs du projet faisaient leur code de leur côté, dans un répertoire à part, du lundi au jeudi.

Et le vendredi, tout le monde se réunissait pour rassembler tous les bouts de codes produits dans leur répertoire,

les mettre en commun et résoudre d'éventuels bugs arrivés lors de cette mise en commun

on passait 1/5 de notre temps à mettre en commun nos codes

Réservation de fichiers

On ne sépare pas le projet en ayant 1 répertoire par développeur. On bosse tous sur le même répertoire, les mêmes fichiers sur un lecteur réseau (un répertoire sur une machine dans le réseau de l'entreprise, qui est partagé avec tous les développeurs).

Mais si on est 2 à modifier un seul fichier, alors les modifs de l'un écrasent les modifs de l'autre, et c'est la guerre.

Donc, dans l'open space, quand on code sur un fichier, on le dit à tous les autres. On se réservait ainsi l'accès au fichier…

les développeurs doivent être dans le même bureau et il persiste des risques d'écrasement de modifs de collègues, sans moyen de les récupérer

Concept de branches

Une branche pourrait être comparée à la copie du répertoire du projet, dans laquelle on souhaite travailler, en tant que développeur.

Mais cela signifierait qu'on aurait 2 répertoires, le répertoire du projet, et la copie dans laquelle on travaille.

Or, tout l'intérêt de git (et d'autres systèmes de versionning) est justement de toujours travailler sur un seul et unique répertoire de projet sur notre disque dur.

Mais alors comment git permet d'avoir plusieurs versions différentes, dans un seul répertoire ?

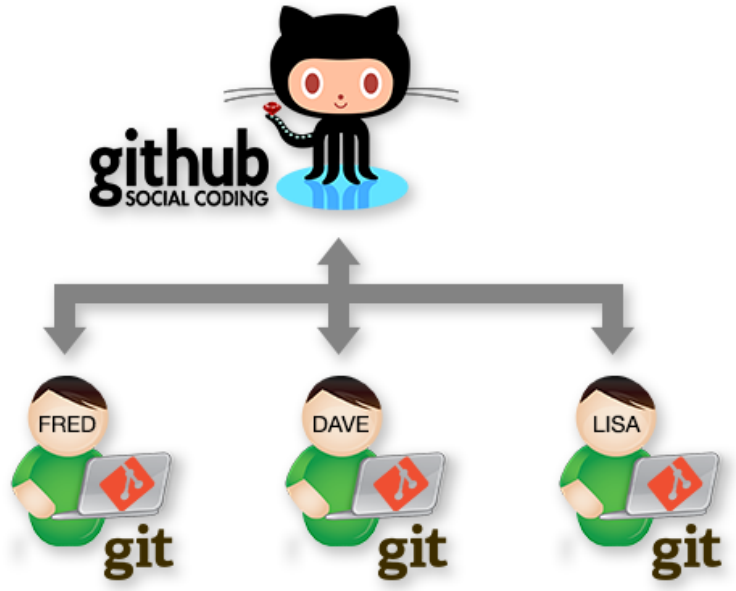
- git, une fois activé dans le répertoire du projet, va pouvoir modifier les répertoires et fichiers
- pour chaque commit, git sauvegarde uniquement les différences, ligne par ligne
- ensuite, il se base sur ces lignes ajoutées/supprimées pour recréer les fichiers à un commit précis, sur la branche actuelle

Lorsqu'on change de branche, git fait le même calcul pour fournir les fichiers dans la version de la branche demandée.

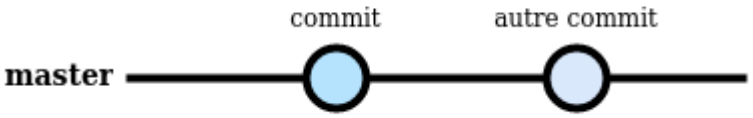
En résumé

- la branche master est la branche principale de développement.
- chaque commit (sauvegarde) est liée à une seule et unique branche.
- donc je peux faire des commits sur une branche, sans que la branche master soit impactée.
- et une fois mon travail fini, je peux « fusionner » ma branche dans la branche master. Et là, c'est git qui s'occupe de mélanger les 2 codes
  (en fait, il va tenter de récupérer les commits spécifiques à la branche dans la branche master).
- il y arrive très souvent sans aide, mais il faudra parfois l'aider (conflits).

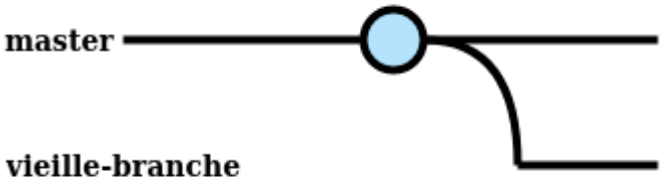
Exemple de workflow



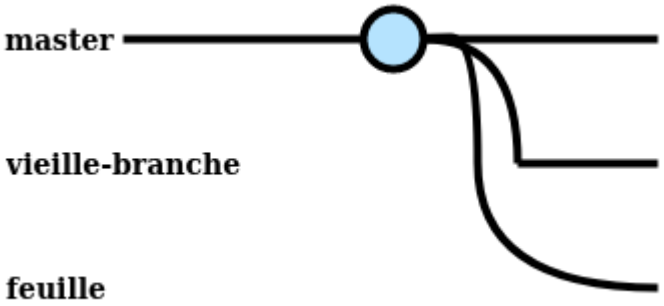
- notre projet n'a qu'une seule branche master, avec tout le code source



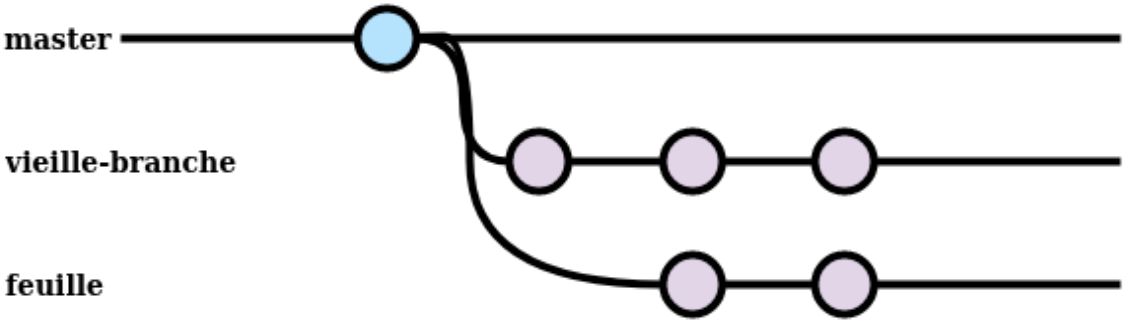
- *Dave* doit créer une fonctionnalité, il crée alors une branche vieille-branche, reprenant le code de la branche master. Il peut alors coder et effectuer des commits, autant qu'il le souhaite dans sa branche



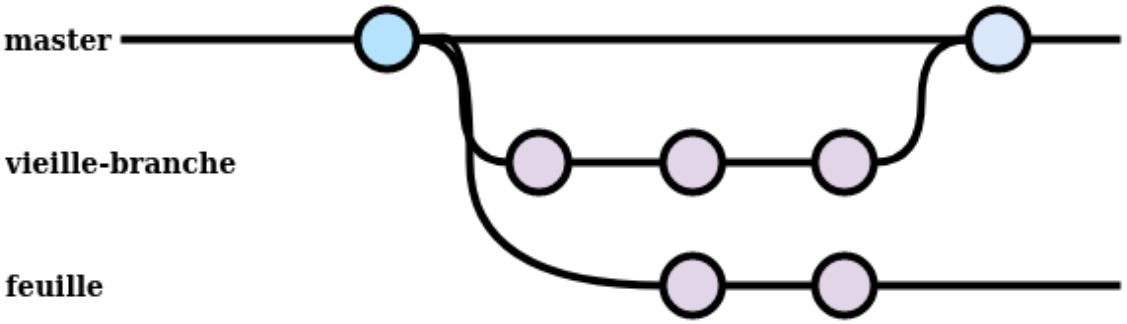
- *Lisa*, elle aussi, doit créer une fonctionnalité (une autre), elle fait comme *Dave*, et crée une branche nommée feuille



- pendant leur développement, *Lisa* et *Dave* peuvent faire des commits comme ils l'entendent, travailler sur les mêmes « fichiers », ils ne gênent pas leurs collègues

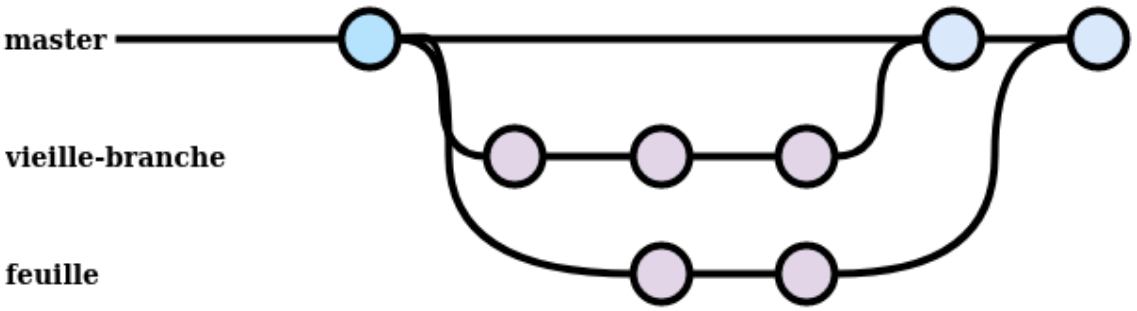


- après 1 jour, *Dave* a terminé sa fonctionnalité
  - il a régulièrement fait des commits de son code
  - il récupère la dernière version de la branche master (mais il n'y a eu aucun changement)
  - il fusionne alors sa branche vieille-branche avec la branche master
  - aucun conflit, c'est parfait



  - il peut alors supprimer sa branche vieille-branche, elle n'est plus utile

- 2 jours plus tard, *Lisa* a terminé aussi sa fonctionnalité
    - elle a régulièrement fait des commits de son code
    - elle récupère la dernière version de la branche master (elle récupère ainsi les changements de *Dave*)
    - elle fusionne alors sa branche feuille avec la branche master
    - aucun conflit, c'est parfait



    - elle peut alors supprimer sa branche feuille, elle n'est plus utile
- les 2 fonctionnalités sont en place

Merci qui ?

Il fait quoi Fred dans cette histoire ?

Les commandes

Lister toutes les branches

```
git branch -a
```

Git affichera alors les branches locales (ta machine) et distantes (GitHub).

De plus, un * indiquera la branche courante.

Créer une branche

```
git checkout -b nom-de-la-branche
```

Git va créer une nouvelle branche, avec le code source de la branche sur laquelle on se trouve au moment de l'exécution de la commande.

Changer de branche

```
git checkout nom-de-la-branche
```

Faire un push d'une branche

```
git push
```

Attention, si la branche existe en local (ta machine) mais pas sur le remote (origin / GitHub), il faudra alors exécuter la commande ci-dessous.

Faire un push d'une nouvelle branche

```
git push -u origin nom-de-la-branche
```

A ne faire qu'une seule fois par branche

Fusionner une branche

1. se placer sur la branche de destination, disons master
2. `git checkout master`

3. récupérer les dernières modifications effectuées sur la branche de destination (master)
4. `git pull`
5. demander la fusion de la branche feuille dans la branche courante (master)
6. `git merge feuille`
7. faire un push sur origin de cette fusion
8. `git push`