Api/casoft Init - Jour 4 - Git

Rémy Huet
Thibaud Duhautbout
Quentin Duchemin
Romain Maliach

Association Picasoft

Jeudi 24 janvier 2019

picasoft

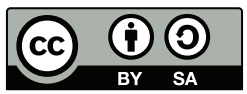


Table des matières

Qu'est-ce que Git ?

À propos

Git est un **logiciel** de **gestion de version**. Il est **open source** et publié sous **licence libre**

Sur quelles plateformes ?

Git est aussi bien disponible sur les distributions **GNU/Linux** que sur **MacOS** et sur **Windows**. On trouve également des applications Git pour **Android**.

Pourquoi la gestion de version ?

- Sauvegarde **incrémentale** du travail
- **Suivi** des modifications et **retour en arrière**
- **Partage** des modifications
- **Centralisation** des sources
- **Collaboration** simplifiée
- Possibilité de **maintenir plusieurs versions**

Différents logiciels de gestion de version

Et plein d'autres !

37 systèmes recensés sur Wikipedia

(`https://en.wikipedia.org/wiki/Comparison_of_version_control_software`)

Petite histoire de Git

- Créé en 2005 par les développeurs du noyeau Linux
- Système de gestion de version distribué
- Rapide
- Possibilité de développements non-linéaires (branches)
- Popularité grandissante chez les développeurs

Configuration de git et du dépôt

On commence par créer un nouveau répertoire de travail qu'on va suivre avec Git :

```
$ mkdir git-example
$ cd git-example
```

On initialise ensuite le dépôt Git dans ce répertoire :

```
$ git init
Dépôt Git vide initialisé dans /home/user/git-example/.git
```

Cette commande crée un répertoire caché `.git` dans le dossier courant, qui contient toutes les informations nécessaires pour enregistrer les versions.

Le contenu de ce répertoire ne sera pas détaillé dans cette formation, ne pas le modifier à la main si vous ne savez pas ce que vous faites !

À partir de maintenant, Git prend en charge la gestion de version du répertoire courant.

Configurer son identité

« Comment Git sait qui fait quoi sur le projet ? »

Il faut commencer par configurer son **identité**, qui sera associée à chaque mise à jour. Elle est définie par un **nom** et une **adresse mail**.

Il y a deux possibilités de configuration de l'identité avec Git :

- *configuration globale*, associée à l'utilisateur et valable pour tous les projets Git de cet utilisateur
- *configuration locale*, utilisée uniquement dans le répertoire courant

Configurer son identité – application

En pratique, on utilise la commande `git config` avec une option pour préciser si on configure une identité locale ou globale :

Pour une configuration globale :

```
$ git config --global user.name "<prénom nom>"
$ git config --global user.email "<adresse email>"
```

Pour une configuration locale :

```
$ git config --local user.name "<prénom nom>"
$ git config --local user.email "<adresse email>"
```

Fonctionnement de Git

Les versions sont enregistrées par Git sous forme de *commits*.

Pour faire court

Un commit est un **point de sauvegarde**

À quoi ça sert ?

- Les commits sont enregistrés les uns à la suite des autres ;
- Sauvegarde incrémentale des modifications ;
- Possibilité de revenir à une version donnée.

Git s'organise en trois espaces principaux qui enregistrent les différents états du travail (cf slide suivante).

Working Directory vs. Staging area vs. Repository

Explications

Repository [dépôt]

Le Repository (ou dépôt) correspond aux fichiers dans l'état de la dernière validation connue par git.

Working Directory [répertoire de travail]

Le Working Directory correspond à l'état actuel du répertoire git :

- nouveaux fichiers pas encore ajoutés (*suivis*) au Repository ;
- fichiers modifiés depuis la dernière version.

Staging Area [zone de préparation / zone d'index]

Zone intermédiaire entre le Working Directory et le Repository. Elle contient les modifications apportées dans le Working Directory que git va ajouter au Repository.

Créer des versions

Working Directory <-> Staging Area

Pour créer une version, il faut d'abord préarer le commit : les modifications apportées au *working directory* sont ajoutées au *staging area*.

Ajouter les modifications d'un fichier pour validation :

```
$ git add <fichier(s)>
```

Ajouter toutes les modifications pour validation (tous les fichiers) :

```
$ git add -A
```

Enlever les modifications d'un fichier de la validation :

```
$ git reset <fichier>
```

(ne change pas le contenu du fichier mais indique juste à git d'ignorer ses modifications pour la validation)

```
$ echo "Je suis le premier fichier utilisé pour cette API sur git" > API.txt

$ git status
Sur la branche master
Aucun commit
Fichiers non suivis:

        API.txt

$ git add API.txt

$ git status
Sur la branche master

Aucun commit

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)

        nouveau fichier : API.txt

$ git reset API.txt

$ git status
Sur la branche master
Aucun commit
Fichiers non suivis:

        API.txt

$ git add API.txt
```

Créer des versions

Staging Area <-> Repository

Une fois que la version est prête, on peut valider le commit pour envoyer la version du *staging area* vers le *repository*.

Pour valider les changements qui ont été ajoutés au staging area :

```
$ git commit
```

Le commit doit contenir un message qui résume le contenu des modifications. Pour l'entrer directement :

```
$ git commit -m "<message>"
```

```
$ git commit -m "Ajout du premier fichier"
[master (commit racine) 712951d] Ajout du premier fichier
 1 file changed, 1 insertion(+)
 create mode 100644 API.txt
```

Dissection d'un commit

À propos du commit

- Chaque commit possède un identifiant unique ;
- Un commit est asocié à une unique personne ;
- L'historique des commits est incrémental. Tout commit (excepté le premier) a un commit « père » ;
- Un commit correspond à une version figée du projet ;
- On peut naviguer dans les commits.

Visualiser des différences

Git log

Afficher l'historique des commits

```
$ git log
commit 712951dbb3ee0cc4d248582dc5408da3afdec853 ( HEAD -> master )
Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
Date:   Thu Jan 3 15:37:54 2019 +0100

            Ajout du premier fichier
```

On peut voir ici :

- L'identifiant unique du commit ;
- L'auteur (et son mail) ;
- La date du commit ;
- Le message qui a été mis lors du commit.

Visualiser des différences

Git diff

Le diff permet de voir les modifications apportées au dépôt :

- Depuis l'état du staging area : `$ git diff` ;
- Depuis le dernier commit : `$ git diff HEAD` ;
- Depuis un commit quelconque : `$ git diff <id_commit>` ;
- Entre deux commits quelconques :
  `$ git diff <id_commit_départ> <id_commit_arrivée>`.

```
$ echo "J'ajoute une ligne à mon fichier" >> API.txt

$  git diff

 diff --git a/API.txt b/API.txt
 index 7eec119..5596950 100644
 --- a/API.txt
 +++ b/API.txt
@@ -1 +1,2 @@
 Je suis le premier fichier utilisé pour cette API sur git
+J'ajoute une ligne à mon fichier

$  git add API.txt

$  git diff

$  git diff HEAD

 diff --git a/API.txt b/API.txt
 index 7eec119..5596950 100644
 --- a/API.txt
 +++ b/API.txt
@@ -1 +1,2 @@
 Je suis le premier fichier utilisé pour cette API sur git
+J'ajoute une ligne à mon fichier

$  git commit -m "Second commit"

[master 5b63bd1] Second commit
 1 file changed, 1 insertion(+)
```

```
$ git log

commit 5b63bd1f9afe4685b60074030960e39bb5152e67 (HEAD -> master)
Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
Date:   Thu Jan 3 15:55:33 2019 +0100

    Second commit

commit 712951dbb3ee0cc4d248582dc5408da3afdec853
Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
Date:   Thu Jan 3 15:37:54 2019 +0100

    Ajout du premier fichier

$  git diff 712951 5b63bd

 diff --git a/API.txt b/API.txt
 index 7eec119..5596950 100644
 --- a/API.txt
 +++ b/API.txt
@@ -1 +1,2 @@
 Je suis le premier fichier utilisé pour cette API sur git
+J'ajoute une ligne à mon fichier
```

Visualiser des différences

Git show

Pour visualiser les modifications introduites par un commit spécifique :

`$ git show <sha du commit>`

```
$ git show 5b63b

commit 5b63bd1f9afe4685b60074030960e39bb5152e67 (HEAD -> master)
Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
Date:   Thu Jan 3 15:55:33 2019 +0100

    Second commit

diff --git a/API.txt b/API.txt
index 7eec119..5596950 100644
--- a/API.txt
+++ b/API.txt
@@ -1 +1,2 @@
 Je suis le premier fichier utilisé pour cette API sur git
+J'ajoute une ligne à mon fichier
```

Qu'est-ce que le HEAD ?

Le HEAD est l'*état courant* du repository. On peut le voir comme le commit sur lequel « on se situe ».

```
$ git log --graph --decorate
* commit 5b63bd1f9afe4685b60074030960e39bb5152e67 ( HEAD -> master)
| Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
| Date:   Thu Jan 3 15:55:33 2019 +0100
|
|     Second commit
|
* commit 712951dbb3ee0cc4d248582dc5408da3afdec853
  Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
  Date:   Thu Jan 3 15:37:54 2019 +0100

      Ajout du premier fichier
```

Pointer avant le HEAD

HEAD est une **etiquette** sur un commit.
Pour pointer sur le commit avant HEAD, on peut utiliser `HEAD~1` *(~2 pour deux commits . . .)*

Git tag

Qu'est ce qu'un tag ?

Un tag est **une etiquette** sur un commit. Il permet par exemple de marquer une version particulière du projet (une *release*)

```
$ git tag mon_tag HEAD~1
$ git log --graph --decorate
* commit 5b63bd1f9afe4685b60074030960e39bb5152e67 (HEAD -> master)
| Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
| Date:   Thu Jan 3 15:55:33 2019 +0100
|
|     Second commit
|
* commit 712951dbb3ee0cc4d248582dc5408da3afdec853 (tag: mon_tag)
  Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
  Date:   Thu Jan 3 15:37:54 2019 +0100

      Ajout du premier fichier
```

Enregistrer les modifications locales

« Je peux avoir la dernière version du projet ?
– Euh, là non j'ai fait des modifications qui ne fonctionnent pas . . .
– Fais un git stash ! »

`git stash` permet de **mettre de côté** des modifications non validées.

```
$ echo 'Encore une ligne en plus !' >> API.txt
$ cat API.txt
Je suis le premier fichier utilisé pour cette API sur git
J'ajoute une ligne à mon fichier
Encore une ligne en plus !
$ git stash
Copie de travail et état de l'index sauvegardés dans WIP on master: 5b63bd1 Second commit
$ cat API.txt
Je suis le premier fichier utilisé pour cette API sur git
J'ajoute une ligne à mon fichier
```

Restaurer les modifications locales

« Et comment je les récupère maintenant ?
– Facile ! git stash pop ! »

`git stash pop` permet de réappliquer les modifications qui avaient été mises de côté.

```
$ git stash pop
Sur la branche master
Votre branche est à jour avec 'origin/master'.

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

  modifié :         API.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
refs/stash@0 supprimé (c0c5cd92988e5f78a8c929e7fd3bb34bb1067fda)

$ git commit -am ''Troisième commit''
[master a04da65] Troisième commit
 1 file changed, 1 insertion(+)
```

Changer de version

« Dis, la prof nous demande la version de la semaine dernière, comment je peux la récupérer ? – Utilises git checkout ! »

`git checkout` permet de déplacer le `HEAD` sur un commit via son sha ou une etiquette (un tag par exemple)

```
$ git log --graph --decorate
* commit a04da653083b6b0ba3eea2bce98d903acfd0a4d3 (HEAD -> master)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 10:16:10 2019 +0100
|
|     Troisième commit
|
* commit 5b63bd1f9afe4685b60074030960e39bb5152e67 <---- Cette version là !
| Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
| Date:   Thu Jan 3 15:55:33 2019 +0100
|
|     Second commit
|
* commit 712951dbb3ee0cc4d248582dc5408da3afdec853 (tag: mon_tag)
  Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
  Date:   Thu Jan 3 15:37:54 2019 +0100

      Ajout du premier fichier
```

```
$ git checkout HEAD~1
Note : extraction de 'HEAD~1'.

Vous êtes dans l'état « HEAD détachée ». Vous pouvez visiter, faire des modifications
expérimentales et les valider. Il vous suffit de faire une autre extraction pour
abandonner les commits que vous faites dans cet état sans impacter les autres branches

Si vous voulez créer une nouvelle branche pour conserver les commits que vous créez,
il vous suffit d'utiliser « checkout -b » (maintenant ou plus tard) comme ceci :

  git checkout -b <nom-de-la-nouvelle-branche>

HEAD est maintenant sur 5b63bd1 Second commit

$ git status
HEAD détachée sur 5b63bd1
rien à valider, la copie de travail est propre

$ git log --graph --decorate --all
* commit a04da653083b6b0ba3eea2bce98d903acfd0a4d3 (master)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 10:16:10 2019 +0100
|
|     Troisième commit
|
* commit 5b63bd1f9afe4685b60074030960e39bb5152e67 (HEAD)
| Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
| Date:   Thu Jan 3 15:55:33 2019 +0100
|
|     Second commit
|
[...]
```

Revenir au dernier commit

« Et comment je retourne à ma version ? »

```
$ git checkout master
La position précédente de HEAD était sur 5b63bd1 Second commit
Basculement sur la branche 'master'
Votre branche est à jour avec 'origin/master'.

$ git log --graph --decorate
* commit a04da653083b6b0ba3eea2bce98d903acfd0a4d3 (HEAD -> master)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 10:16:10 2019 +0100
|
|     Troisième commit
|
* commit 5b63bd1f9afe4685b60074030960e39bb5152e67
| Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
| Date:   Thu Jan 3 15:55:33 2019 +0100
|
|     Second commit
|
* commit 712951dbb3ee0cc4d248582dc5408da3afdec853 (tag: mon_tag)
  Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
  Date:   Thu Jan 3 15:37:54 2019 +0100

      Ajout du premier fichier
```

Annuler des changements

« Dis, j'ai fait des modifications qui ont tout cassé, y'a moyen de revenir en arrière ?
– Oui ! git checkout -- »

`git checkout -- <fichier/dossier>` permet de remettre un fichier ou un dossier dans l'état du dernier commit

```
$ echo 'Ligne qui casse tout' | cat | cat | cat > API.txt
$ git diff
diff --git a/API.txt b/API.txt
index a03f78c..85468b3 100644
--- a/API.txt
+++ b/API.txt
@@ -1,3 +1 @@
-Je suis le premier fichier utilisé pour cette API sur git
-J'ajoute une ligne à mon fichier
-Encore une ligne en plus !
+Ligne qui casse tout

$ git checkout -- API.txt

$ git diff
```

Sans modification d'historique

« Dis, je me suis planté sur un commit, j'aimerais bien l'annuler . . . C'est possible ?
Oui ! Utilises git revert ! »

`git revert` permet d'annuler un commit, en en créant un commit qui fait les modifications inverses.

```
$ git revert HEAD
[master 6e23b65] Revert "Troisième commit"
 1 file changed, 1 deletion(-)
```

```
$ git show HEAD

commit 6e23b65243628f81924f650466c86ef3a8e42ec5 (HEAD -> master)
Author: huetremy <remy.huet@etu.utc.fr>
Date:   Fri Jan 11 14:20:59 2019 +0100

    Revert "Troisième commit"

    This reverts commit a04da653083b6b0ba3eea2bce98d903acfd0a4d3.

diff --git a/API.txt b/API.txt
index a03f78c..5596950 100644
--- a/API.txt
+++ b/API.txt
@@ -1,3 +1,2 @@
 Je suis le premier fichier utilisé pour cette API sur git
 J'ajoute une ligne à mon fichier
 -Encore une ligne en plus !

 $ git diff HEAD~2
```

Avec modification d'historique

Git reset

À quoi ça sert ?

- `git reset` permet de remettre le HEAD dans un état spécifique, et modifie l'arbre en conséquence (supprime tous les commits après)
- `git reset --soft` laisse le working directory dans l'état dans lequel il était
- `git reset --hard` remet le working directory dans l'état du commit sur lequel on reset.

```
$ git reset --soft HEAD~1

$ cat API.txt
Je suis le premier fichier utilisé pour cette API sur git
J'ajoute une ligne à mon fichier
```

```
$ git log --graph --decorate --all

* commit a04da653083b6b0ba3eea2bce98d903acfd0a4d3 (HEAD -> master)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 10:16:10 2019 +0100
|
|     Troisième commit
|
* commit 5b63bd1f9afe4685b60074030960e39bb5152e67
| Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
| Date:   Thu Jan 3 15:55:33 2019 +0100
|
|     Second commit
|
* commit 712951dbb3ee0cc4d248582dc5408da3afdec853 (tag: mon_tag)
  Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
  Date:   Thu Jan 3 15:37:54 2019 +0100

      Ajout du premier fichier
```

On voit bien grâce qux deux commandes précédentes que le commit a été supprimé mais que les modifications ont été gardées. . . Maintenant :

```
$ git commit -am ''Commit de revert''
[master eae33f2] Commit de revert
 1 file changed, 1 deletion(-)
```

```
$ git reset --hard HEAD~1
HEAD est maintenant à a04da65 Troisième commit

$ cat API.txt
Je suis le premier fichier utilisé pour cette API sur git
J'ajoute une ligne à mon fichier
Encore une ligne en plus !

$ git log --graph --decorate --all
* commit a04da653083b6b0ba3eea2bce98d903acfd0a4d3 (HEAD -> master)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 10:16:10 2019 +0100
|
|     Troisième commit
|
* commit 5b63bd1f9afe4685b60074030960e39bb5152e67
| Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
| Date:   Thu Jan 3 15:55:33 2019 +0100
|
|     Second commit
|
* commit 712951dbb3ee0cc4d248582dc5408da3afdec853 (tag: mon_tag)
  Author: Thibaud Duhautbout <thibaud@duhautbout.ovh>
  Date:   Thu Jan 3 15:37:54 2019 +0100

      Ajout du premier fichier
```

Le commit a bien été effacé, mais cette fois les modifications n'ont pas été gardées

Avec modification d'historique

Git commit –amend

« Dis, j'aimerais faire une toute petite modification sur un commit, c'est possible ?
– Oui ! Utilises git commit --amend »

`git commit --amend` permet de modifier un commit (tant son message que son contenu, son auteur . . .)

```
$ echo "Je suis l'apiinit" >> API.txt && git commit -am "Commit supplémentaire"

$ sed -i "s|apiinit|Api/casoft init|" API.txt

$ git commit -a --amend
[master 168efba] Commit supplémentaire
 Date: Fri Jan 11 14:53:09 2019 +0100
 1 file changed, 1 insertion(+)
```

```
$ git log --graph --decorate

* commit 168efba77dcfd59ba4346fe4a34427b71db75da7 (HEAD -> master)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 14:53:09 2019 +0100
|
|     Commit supplémentaire
|
* commit a04da653083b6b0ba3eea2bce98d903acfd0a4d3
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 10:16:10 2019 +0100
|
|     Troisième commit
|
[...]

$ git show HEAD
commit 168efba77dcfd59ba4346fe4a34427b71db75da7 (HEAD -> master)
Author: huetremy <remy.huet@etu.utc.fr>
Date:   Fri Jan 11 14:53:09 2019 +0100

    Commit supplémentaire

diff --git a/API.txt b/API.txt
index a03f78c..4af702a 100644
--- a/API.txt
+++ b/API.txt
@@ -1,3 +1,4 @@
 Je suis le premier fichier utilisé pour cette API sur git
 J'ajoute une ligne à mon fichier
 Encore une ligne en plus !
+Je suis l'Api/casoft init
```

Le concept des remotes

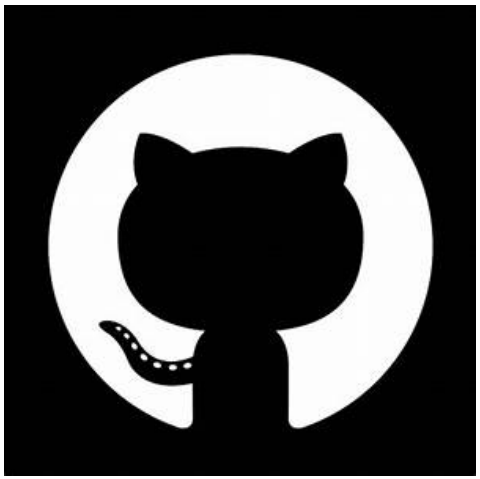
Présentation

Une « remote » est un dépôt git en ligne. Celui-ci se comporte comme un dépôt local, à part qu'il ne possède pas de « working directory ».

Intérêt

- Partage du dépôt git : on peut désormais travailler à plusieurs dessus
- Sauvegarde du travail à distance

Exemples de remotes



Mais surtout : `https://gitlab.utc.fr`

Création d'un dépôt distant

Depuis le gitlab de l'utc

Se rendre sur `https://gitlab.utc.fr`
Une fois connecté avec ses identifiants CAS, il suffit d'appuyer sur le bouton « New project » et de renseigner un nom et un niveau de visibilité

Ajout de clé ssh

Pour simplifier l'authentification au serveur, on ajoutera sa **clé publique ssh** à son profil

Cloner un dépôt distant

Vous avez dit cloner ?

Cloner un dépôt git, c'est récupérer le « repository » distant sur sa machine. Cela fera en même temps la configuration nécessaire pour que le dépôt distant soit « lié » au dépôt local

Et comment on fait ça ?

La commande pour cloner un dépôt est
`git clone <url du repo (https ou ssh)>`

```
$ cd ..

$ git clone git@gitlab.utc.fr:picasoft/apis/h19/init/git
Clonage dans 'git'...
remote: Enumerating objects: 384, done.
remote: Counting objects: 100% (384/384), done.
remote: Compressing objects: 100% (164/164), done.
remote: Total 384 (delta 182), reused 362 (delta 169)
Réception d'objets: 100% (384/384), 399.33 KiB | 1.93 MiB/s, fait.
Résolution des deltas: 100% (182/182), fait.
```

Tirer des changments

Explications

Tirer des changments, c'est **récupérer** des commits depuis la remote.
Pour cela, on utilise tout naturellement la commande `git pull`

```
$ cd git
$ git pull
[...]
```

Pousser des commits

Explications

Pousser un ou plusieurs commits, c'est envoyer du travail local sur le serveur pour le partager/sauvegarder. Cela se fait avec la commande `git push`

```
$ cd ..

$ git clone git@gitlab.utc.fr:<mon_login>/<mon_repo>
[...]

$ cd <mon_repo>

$ touch file && git add -A && git commit -m ''mon premier commit''
[...]

$ git push
Décompte des objets: 3, fait.
Delta compression using up to 8 threads.
Compression des objets: 100% (2/2), fait.
Écriture des objets: 100% (3/3), 939 bytes | 939.00 KiB/s, fait.
Total 3 (delta 1), reused 0 (delta 0)
To gitlab.utc.fr:<mon_login>/<mon_repo>
   e32c9bf..4cf38e2  master -> master
```

Qu'est-ce qu'une forge ?

Forge

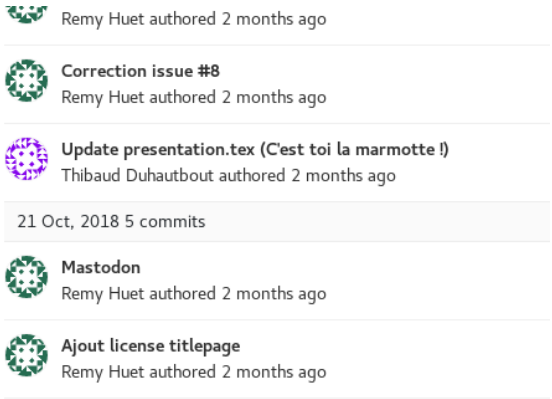
C'est un logiciel complémentaire à Git qui, en plus de gérer les versions, propose :

- L'hébergement de repos
- La gestion des bugs, de la documentation, des tâches à accomplir...
- La protection du repo (mot de passe et clés, droits de l'utilisateur...)
- Un réseau social permettant à une communauté d'interagir avec le repo

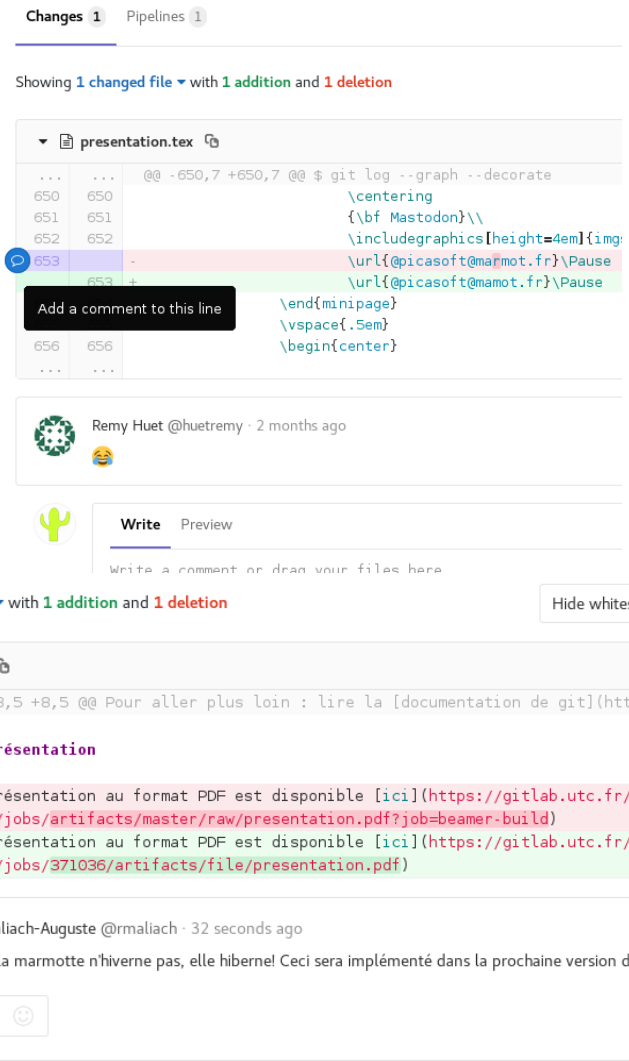
L'UTC héberge une instance de la forge **Gitlab**. Il existe aussi Github, qui est une énorme forge centralisée.

Actions sur des commits

Les commits étant constitués de diffs, on peut commenter ces diffs :
Cliquer sur un commit :



Les diffs s'affichent. Cliquer sur un numéro de ligne et commenter.



Les issues : demander des améliorations

Pour chaque repo, Gitlab tient une liste d'issues. Une issue est un problème ou une demande sur le code. Tout le monde peut déposer des issues mais on peut changer ce comportement.

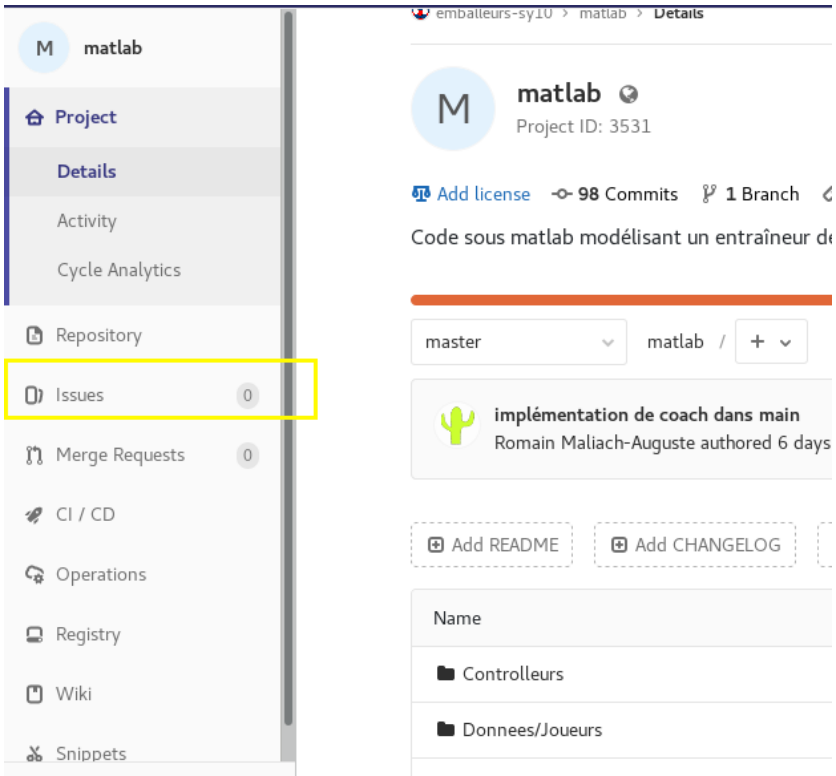
« A quoi ça sert de faire ça sur Gitlab ? Je peux très bien tenir la liste des choses à faire sur mon 3310 ! »

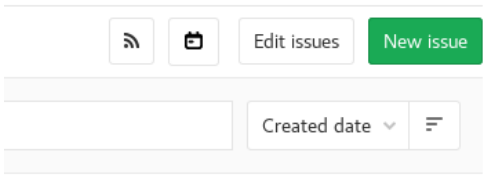
Il y a plusieurs intérêts, par exemple :

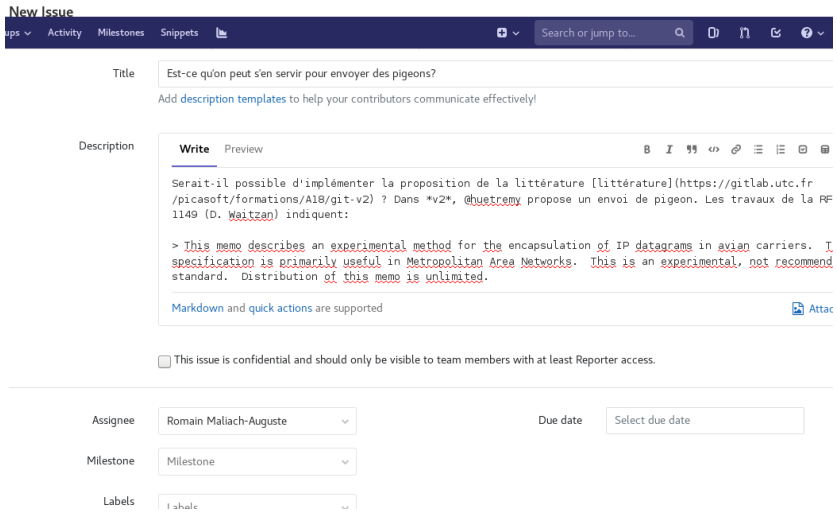
- On peut commenter des fichiers et des diff sans les modifier
- Cela facilite la communication entre les différents développeurs et permet d'assurer un suivi
- On peut assigner (confier) une issue à un développeur
- On peut référencer les issues dans les messages de commit.

Les issues : la pratique

Pour créer une issue, rien de plus simple : on clique sur issues puis sur new.

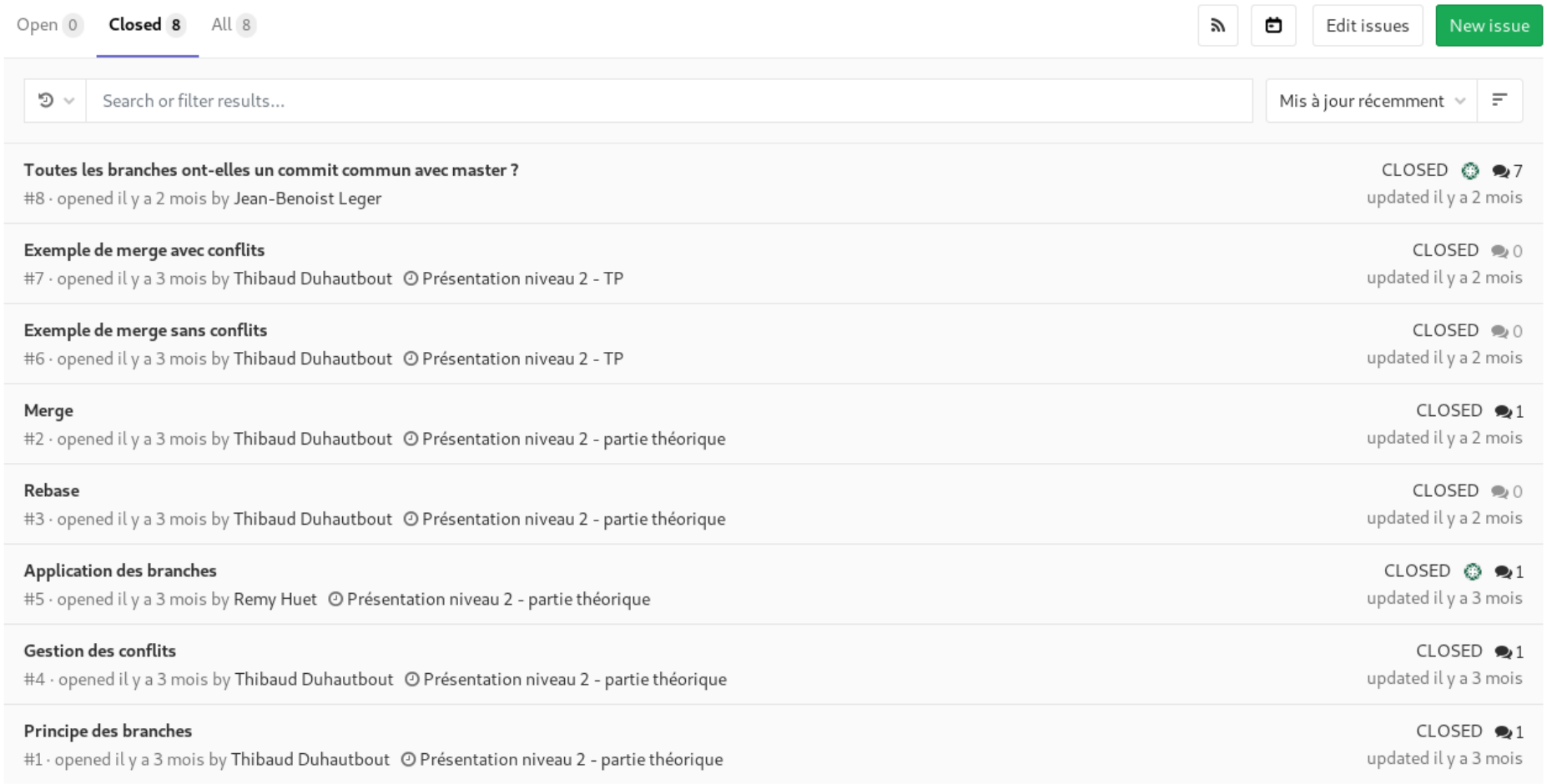






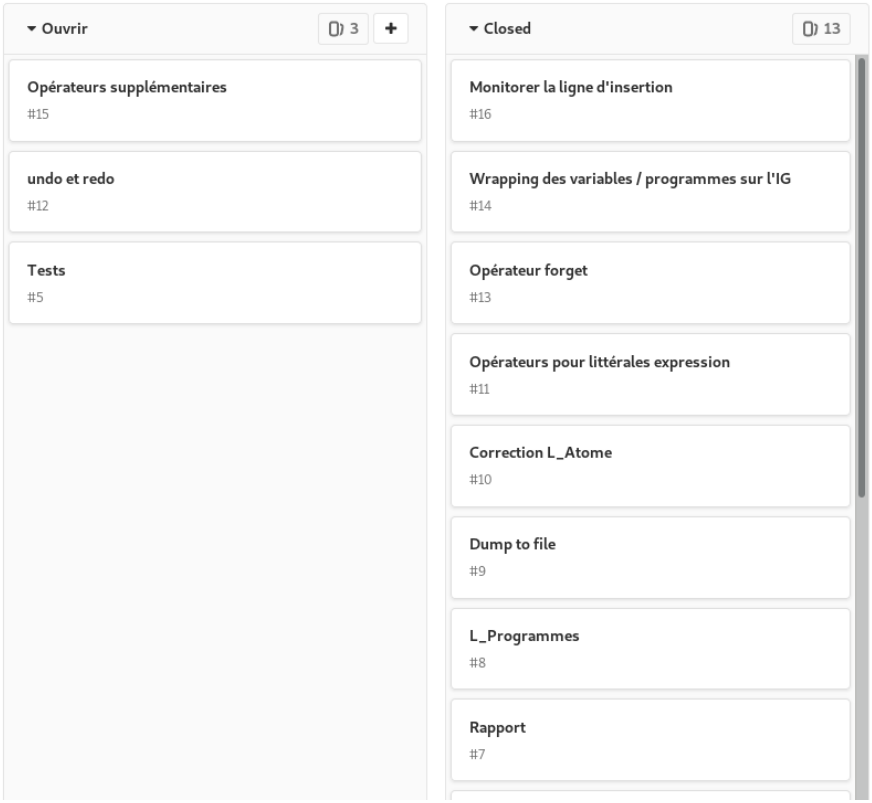
Une méthodologie de travail possible est de lister toutes les tâches à réaliser sous la forme d'issues. On peut aussi s'en servir pour faire remonter les bugs trouvés par les utilisateurs.

Les issues - vision globale



Les issues - tableau

Il est possible de gérer les issues sous une forme de tableau Kanban (ToDoList) :



https://fr.wikipedia.org/wiki/Tableau_kanban

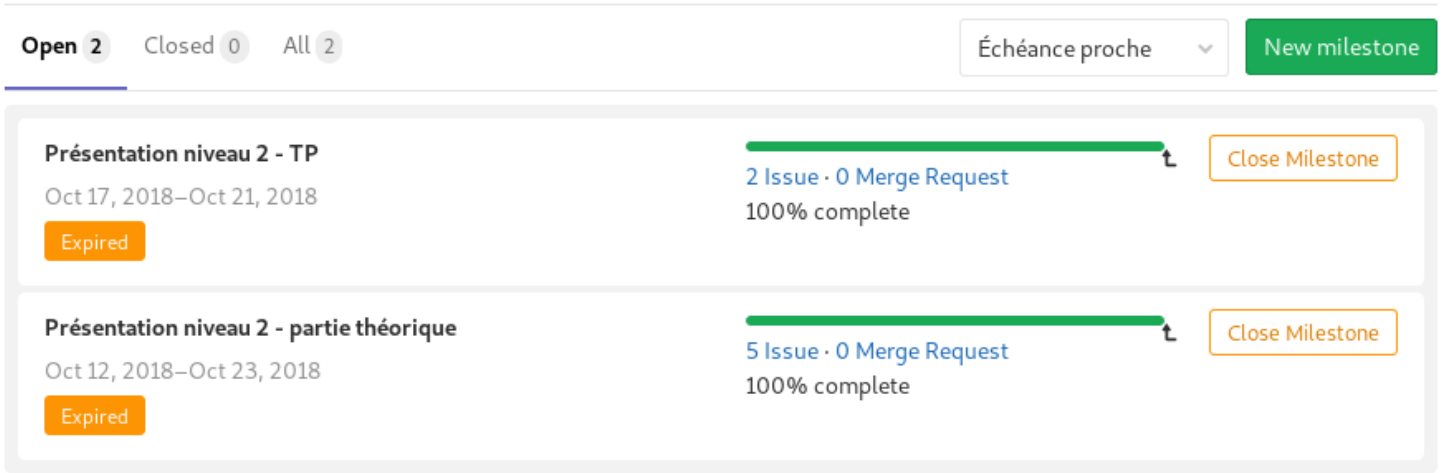
Les issues : gérer son projet

Les issues ne sont pas seulement une façon de remonter des problèmes !
Elles peuvent également être utilisées pour gérer son projet :

- Chaque tâche identifiée est matérialisée par une issue ;
- Les issues sont assignées à chaque collaborateur ;
- Les issues peuvent servir d'espace de discussion sur les points de travail ;
- Une fois les travaux terminés, les issues sont fermées.

Les jalons

Les jalons permettent de regrouper les issues et de visualiser l'avancée globale du projet.



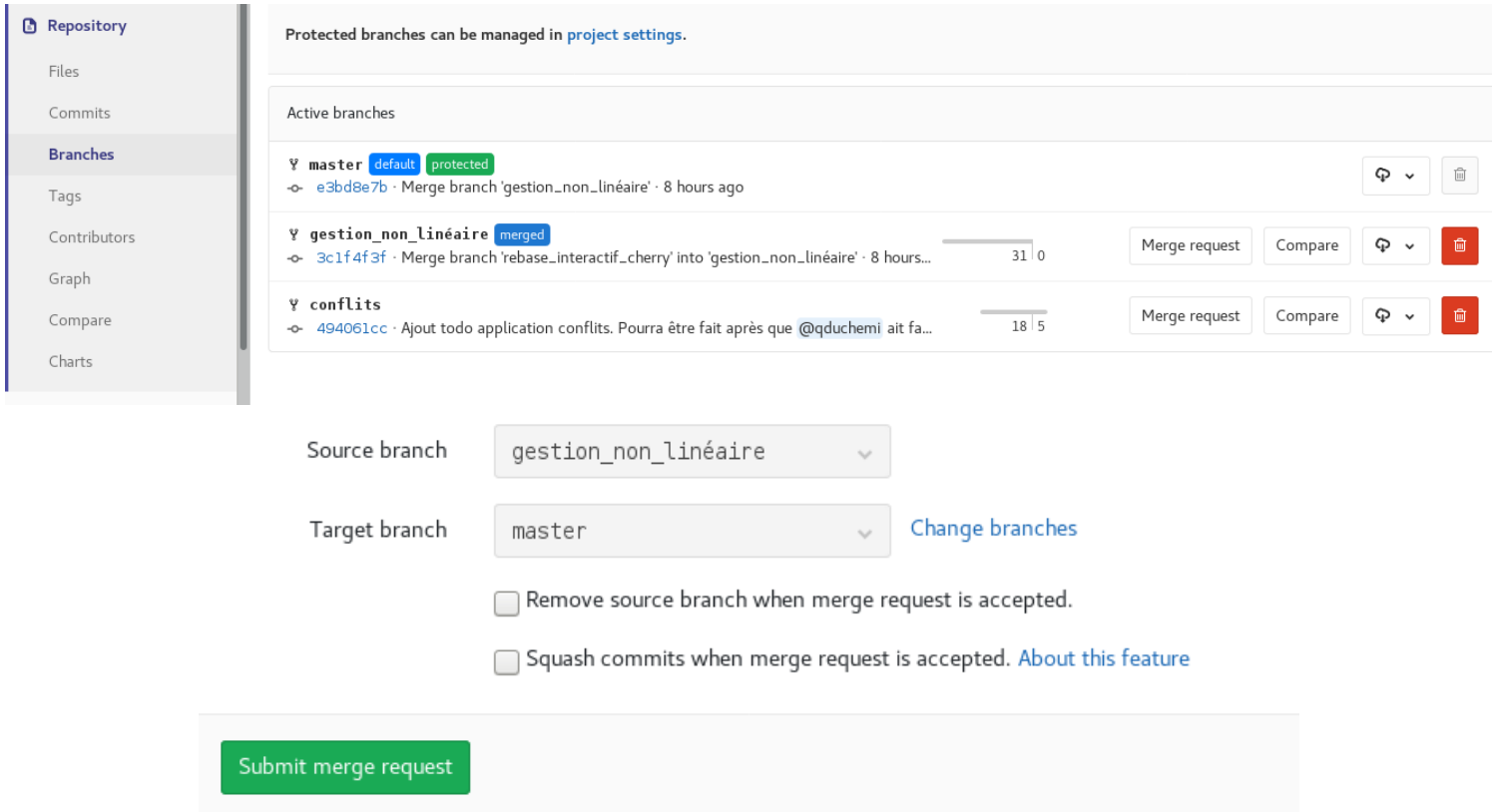
A la fin du jalon, une *merge request* (ça arrive juste après) peut être ouverte.

Les merge request

Les merge request permettent à des développeurs tiers (n'ayant pas d'accès en écriture au repo) de proposer leurs modifications.
Ils peuvent ainsi travailler sur une autre branche (voire une autre remote) puis demander aux propriétaires de merge leurs commits.
Certaines branches peuvent être **protégées** : seuls certains utilisateurs peuvent y écrire.

Les merge request

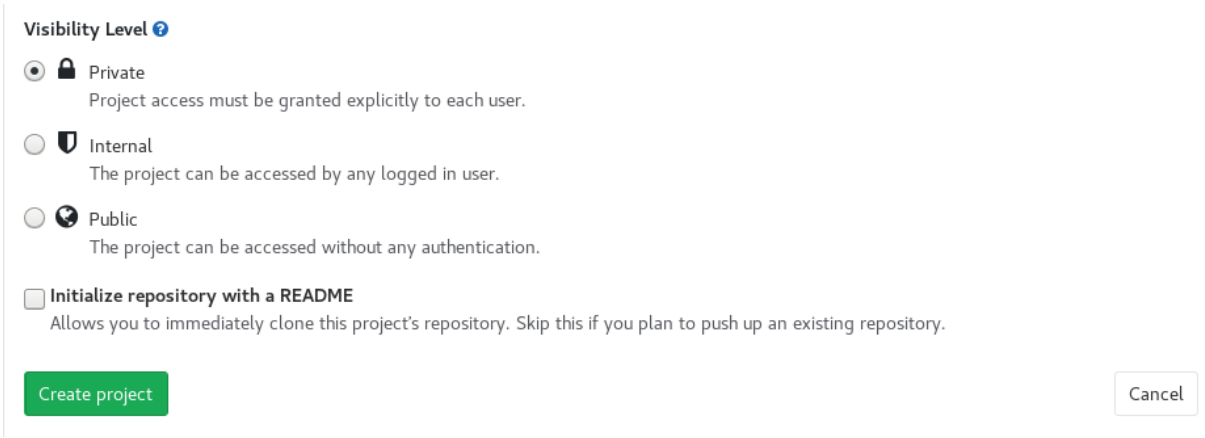
Pour faire une merge request, il suffit de sélectionner les deux branches dans l'interface graphique. Gitlab peut même prédir si le 'git merge' fonctionnera automatiquement ou s'il y aura des conflits.



Gestion des droits

Visibilité

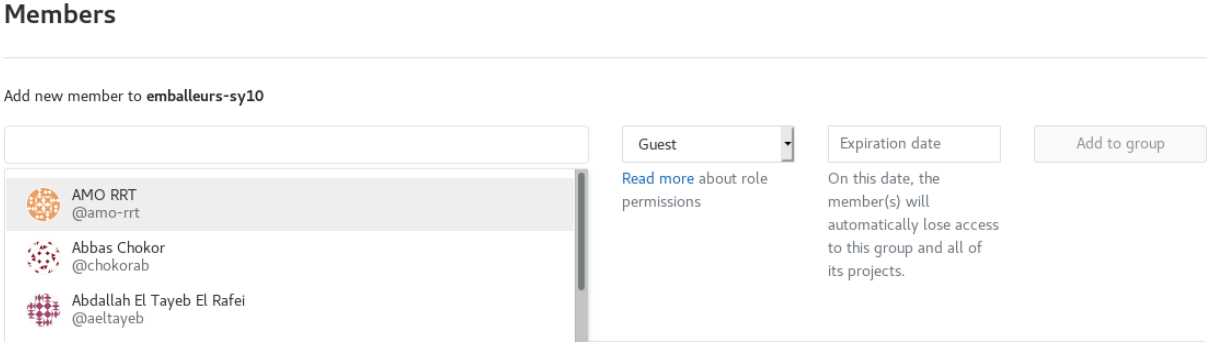
Quand on crée un repo, on choisit sa visibilité. Il y en a 3 sur gitlab : Private, Internal et Public.



Gestion des droits

Rôles

On peut ensuite ajouter des membres habilités à écrire dans le repo via l'onglet members. A l'UTC, on peut trouver des personnes par nom ou login CAS.



Chaque membre a un rôle parmi les suivants :

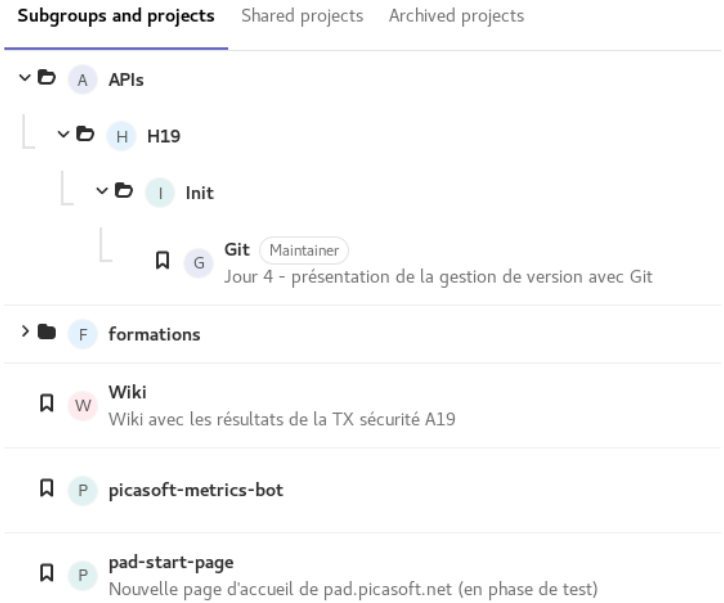
- **Guest** : peut écrire des commentaires et des issues
- **Reporter** : peut lire le code et modérer les commentaires et issues
- **Developer** : peut créer des branches et y push du code.
- **Maintainer** : peut gérer les branches
- **Owner** : toutes les autres permissions

Gestion des droits

Groupes

Les projets gitlab (c'est-à-dire les repos et les infos liées à ces repos) peuvent être rangées dans des groupes et des sous-groupes. Les groupes permettent de représenter des équipes de développeurs, de segmenter leurs droits, d'organiser les repo par thématique.

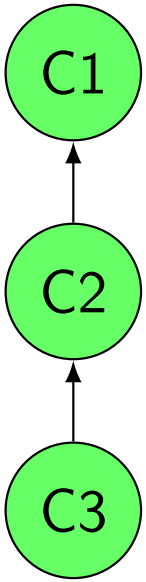
Voici un extrait de l'arborescence de Picasoft :



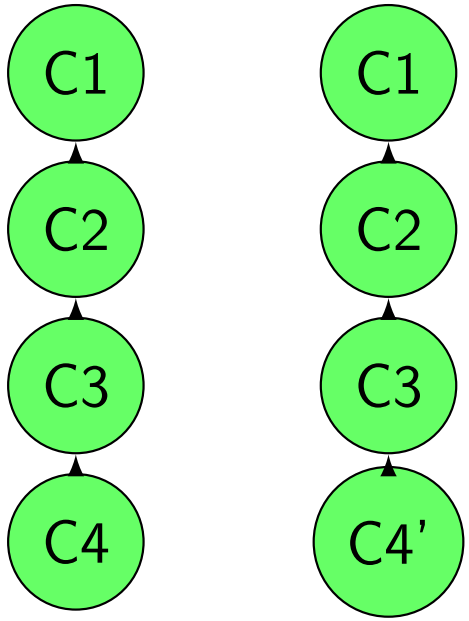
Qu'est-ce qu'un conflit ?

Définition : On parle de conflit lorsque deux personnes on modifié les mêmes lignes d'un fichier **en parallèle** et que git ne peut donc pas savoir quelle version conserver lors d'une fusion.
Concrètement, dans notre utilisation de git, on peut avoir un conflit :

- Lorsque l'on fait un commit sans être à jour avec la branche distante (ce qui impliquera un merge lors du prochain pull) ;
- Quand on ré-applique des modifications qui avaient été mises de côté via un stash.



(a) Situation initiale, alice tire



(b) Alice veut pousser

Figure – Alice a push ses modifications en ayant résolu le conflit

Exemple :

- Alice tire le projet en 1a, il y a 3 commits
- Alice ajoute un commit C4 à sa copie locale du projet
- Pendant ce temps, Bob ajoute son commit C4' sur le projet
- Alice veut pousser mais doit d'abord choisir quelles modifications parmi celles de bob sont pertinentes

Comment se présente un conflit ?

Exemple à ne pas taper

```
$ git merge develop
Fusion automatique de fichier.txt
CONFLIT (contenu) : Conflit de fusion dans fichier.txt
La fusion automatique a échoué ; réglez les conflits et validez le résultat.

$ git status
Sur la branche master
Vous avez des chemins non fusionnés.
  (réglez les conflits puis lancez "git commit")
  (utilisez "git merge --abort" pour annuler la fusion)

Chemins non fusionnés :
  (utilisez "git add <fichier>..." pour marquer comme résolu)

          modifié des deux côtés :  fichier.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")

$ cat fichier.txt
...
<<<<<<< HEAD
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
=======
Integer nec porta orci, non egestas odio.
>>>>>>> develop
...
```

En théorie

- Git fait de la gestion de versions « stupidement ». Il ne comprend pas votre code / votre texte, et ne sait donc pas quoi garder en cas de conflit.
- Résoudre un conflit c'est dire à Git quelle est la bonne verion à garder.
- Il peut s'agit d'une des deux versions uniquement, ou d'un mélange des deux.
- Concrètement cela consite à réécrire les zones délimitées par <<<<<<<<<< et >>>>>>>>>>.

Application

```
$ git merge develop
Fusion automatique de dev.txt
CONFLIT (contenu) : Conflit de fusion dans dev.txt
La fusion automatique a échoué ; réglez les conflits et validez le résultat.

$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.

Vous avez des chemins non fusionnés.
  (réglez les conflits puis lancez "git commit")
  (utilisez "git merge --abort" pour annuler la fusion)

Chemins non fusionnés :
  (utilisez "git add <fichier>..." pour marquer comme résolu)

  modifié des deux côtés :  dev.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")

$ cat dev.txt
Ma première ligne
Resolution de bug
<<<<<<< HEAD
=======
Debut nouvelle fonctionnalité
>>>>>>> develop
```

Application

Pour résoudre le conflit, on va dire à git quelle version du fichier garder. Ici, on gardera la ligne « Début de nouvelle fonctionnalité »

On modifie le fichier dev.txt :

```
Ma première ligne
Resolution de bug
Debut nouvelle fonctionnalité
```

Puis on finit le merge :

```
$ git add dev.txt
$ git commit
```

Quand parler de divergence ?

Définition

On parle de **divergence** lorsque deux **historiques** sont **incompatibles** *(c'est à dire qu'on ne peut pas pousser ou tirer car les historiques ne sont pas cohérents)*
Il faut bien faire la différence entre le **retard** d'un historique sur un autre, qui peut se régler via un `push` ou un `pull` et une divergence.

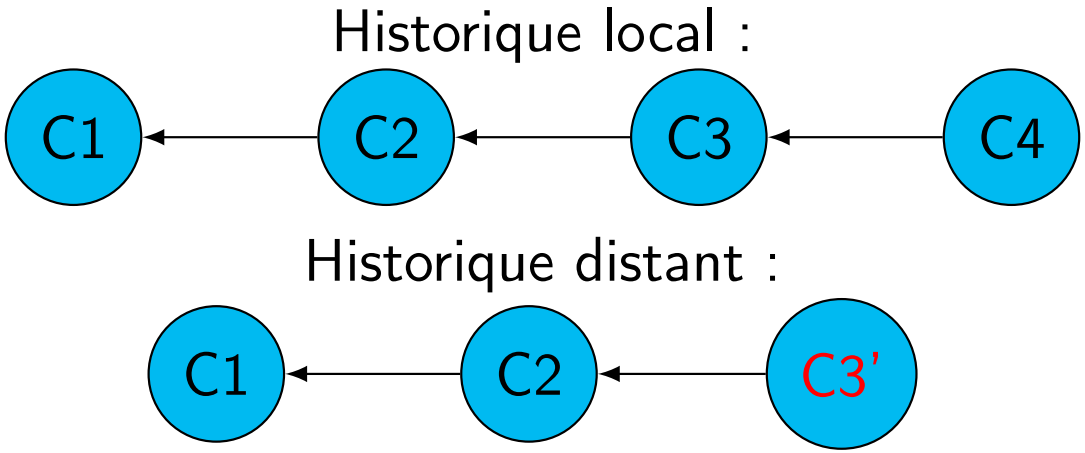


Figure – Exemple de divergence

Comment diverger ?

Explications

On vient de voir qu'une divergence était une incohérence entre deux historiques. On peut donc en déduire plusieurs cas communs de divergence :

- En faisant un commit sans être à jour avec un repo distant
- En ammendant ou supprimant un ou plusieurs commits *via un git commit –amend ou un git reset*
- Lors d'un rebase *(c'est pourquoi on vous a dit de ne jamais rebase une branche déjà suivie)*

Résoudre des divergences ?

Définition

Résoudre une divergence, c'est faire en sorte que les historiques soient cohérents.
Pour ça, on peut :

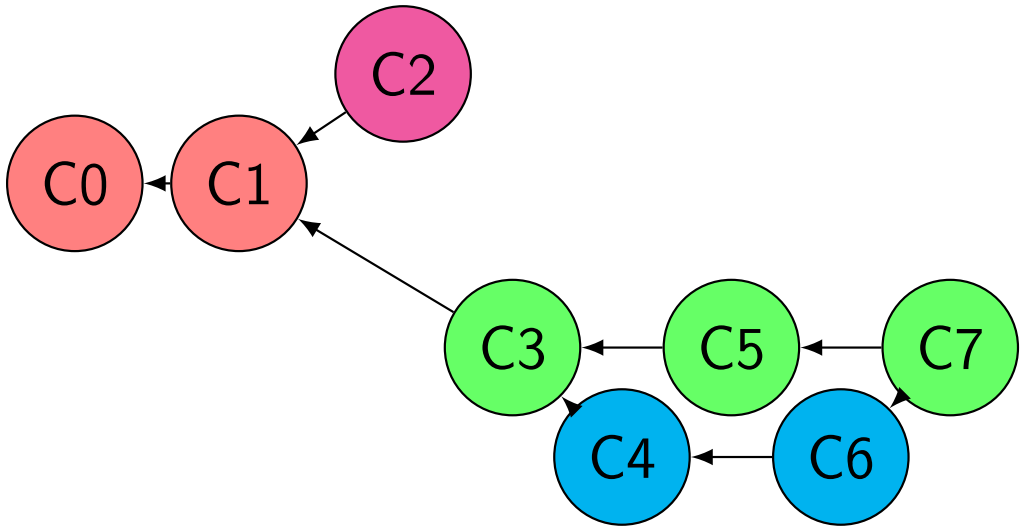
- Remettre toutes les divergences au dessus de l'historique distant via un `git rebase`. Ainsi, si on fait un commit sur master en étant en retard sur le remote, on rebasera master sur origin/master et la divergence sera réglée
- Écrasant son historique local via un `git reset --hard`
- Écrasant l'historique distant via un `git push --force`

Attention

On évitera d'ecraser l'historique distant sans être sûr de soi, car cela créera des divergences avec **tous les autres collaborateurs** du projet.

Principe de la gestion non linéaire

Jusqu'ici, on n'a fait que de la gestion linéaire. Tous les commits étaient sur **une unique branche** (master)
En pratique, tous les commits ne sont pas nécessairement sur la même branche de l'arbre.



Le sens des flèches n'est pas chronologique ! Chaque commit pointe vers son père.

Gestion non linéaire

« Mais pourquoi est-ce qu'on fait ça ? »

Une divergence s'effectue à partir d'un certain point : tout le travail précédent l'instant de divergence est commun à toutes les branches postérieures à la création de la divergence.

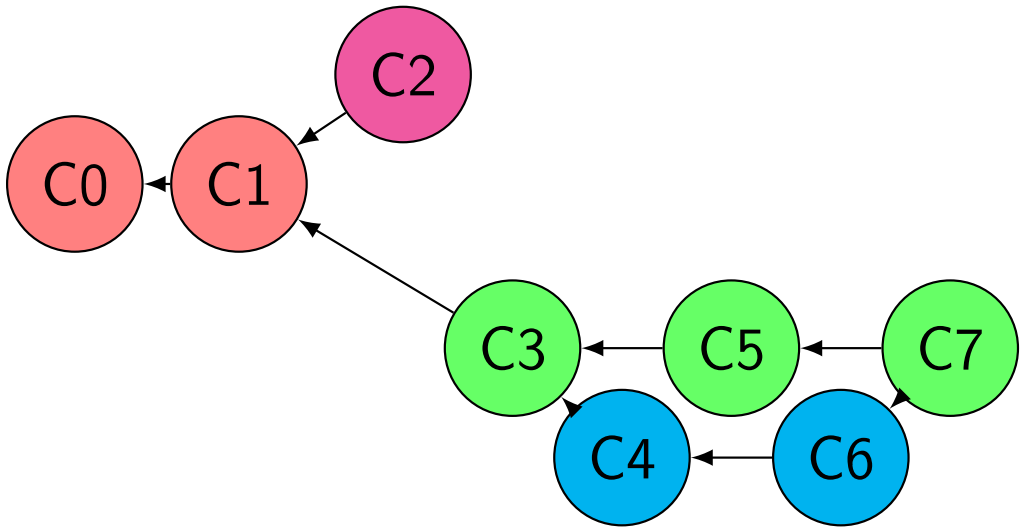
On peut donner quelques raisons « générales » sur l'utilité des divergences :

- isoler les travaux indépendants en cours
- traiter les problèmes d'intégration séparément
- enregistrer des versions spécifiques

En pratique, l'utilisation des divergences dépend beaucoup du type de projet, de la répartition des travaux entre les contributeurs et des méthodes de travail adoptées.

Création d'une divergence

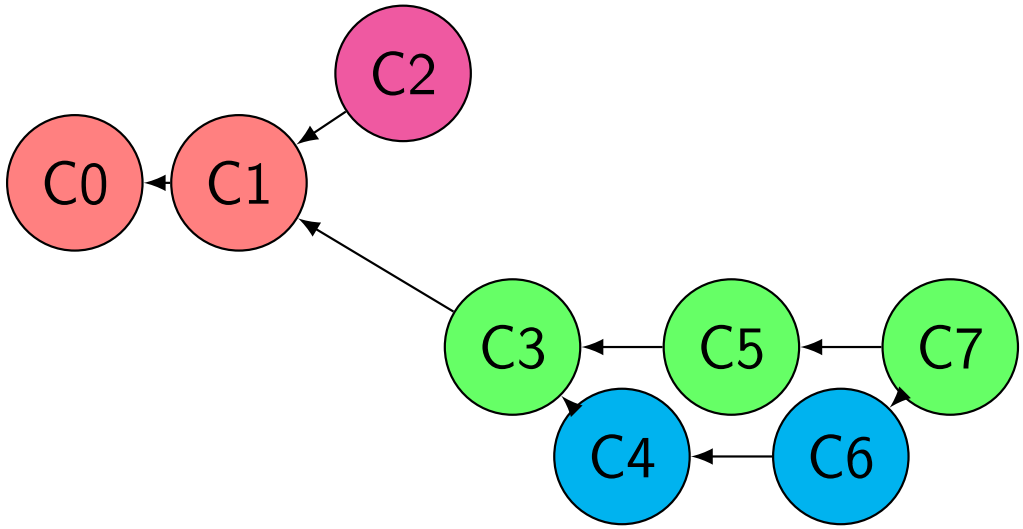
Analyse



- C2 et C3 ont tous les deux C1 comme père
- C2 et C3 introduisent des modifications différentes après C1
- ici, C2 et C3 sont sur **deux branches différentes de C1**

Création d'une divergence

Analyse

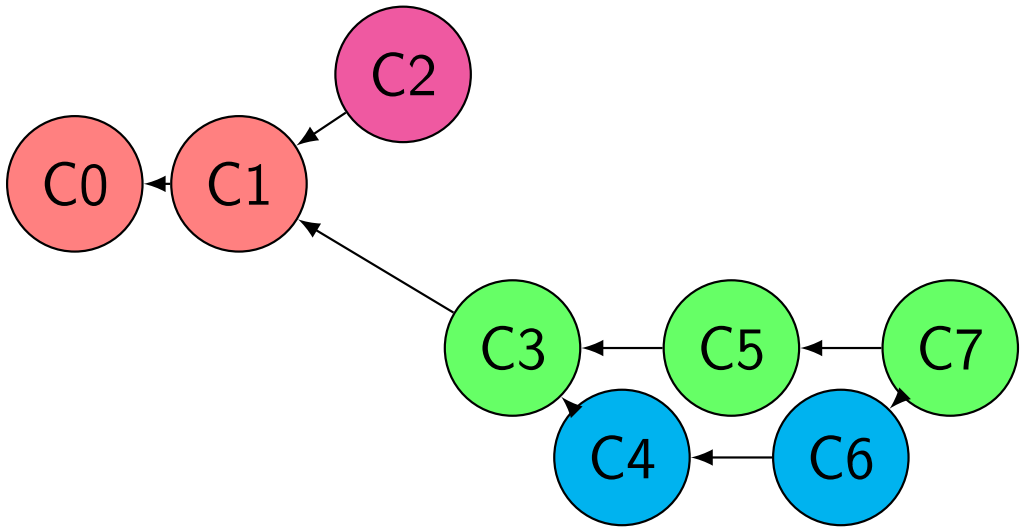


- C4 et C5 ont tous les deux C3 comme père
- C4 et C5 introduisent des modifications différentes après C3
- C4 est sur une branche **différente de C3**
- C5 est sur **la même branche que C3**

Création d'une divergence

Mise en contexte

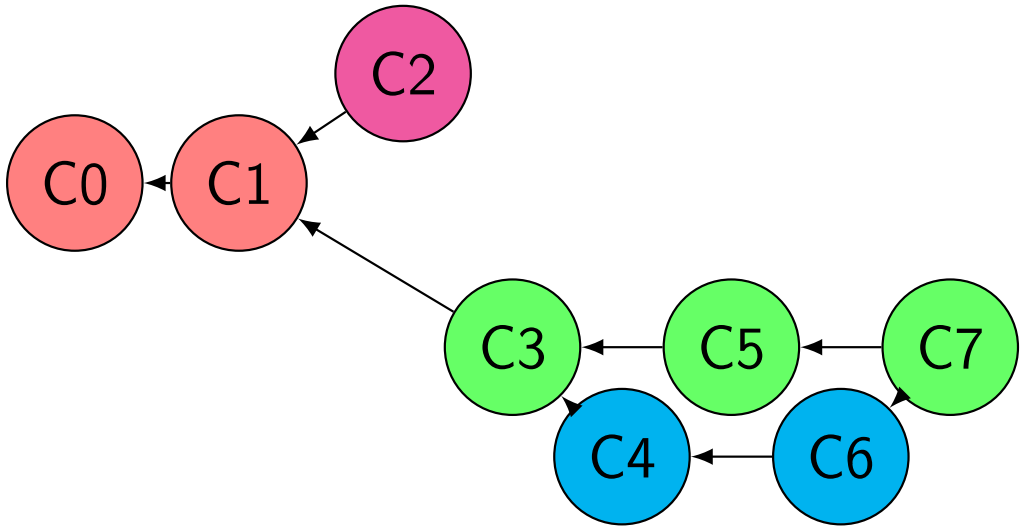
Alice, Bob et Charlie travaillent sur un rapport ensemble et décident d'utiliser Git pour gérer l'avancée de leur travail.



- Alice met en place la structure globale du rapport avec C0 et C1
- Bob se charge de la partie 1, il crée une nouvelle branche dédiée à sa partie et ajoute C2

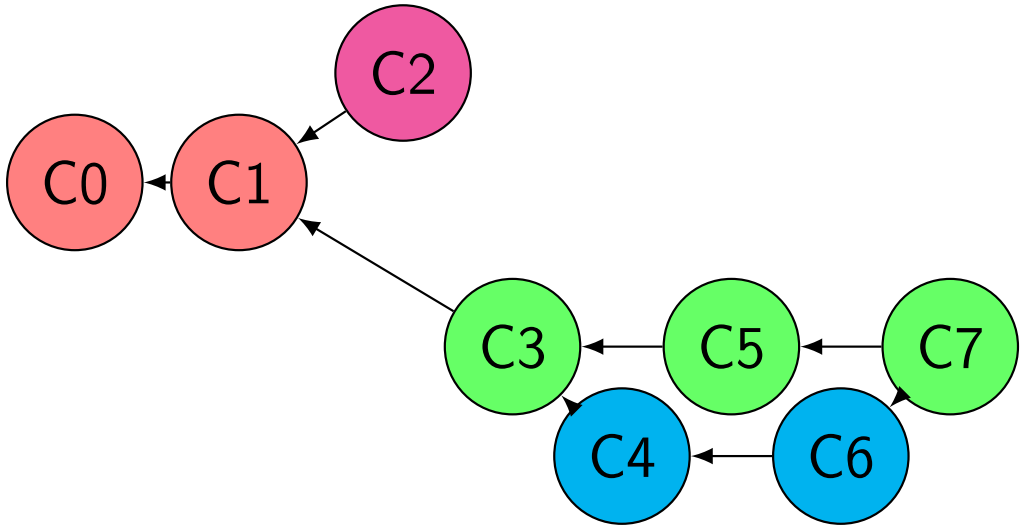
Création d'une divergence

Mise en contexte



- Charlie s'occupe de la partie 2.1 : il ajoute la structure de la partie 2 avec C3 sur une nouvelle branche, et avance sa partie avec C5
- Alice rédige la partie 2.2 : elle crée une nouvelle branche à partir de C3 pour récupérer la structure de la partie 2, puis ajoute C4 et C6

Fusion !



« Attends un peu, et C7 il fait quoi ? Et pourquoi il a deux pères ? »

- C7 est un commit un peu spécial : c'est un **commit de fusion** ;
- Objectif : intégrer les modifications de la branche bleue dans la branche verte (Alice fusionne ses modifications dans la branche de Charlie) ;
- Il a bien **deux** pères (c'est la seule situation où ça arrive) ;
- **Attention !** Si C5 et C6 portent des modifications qui se recouvrent, la fusion va créer des **conflits** qu'il faudra régler avant de créer C7.

Gestion pratique des branches

Création d'une branche

- `git branch <nom>` pour créer une branche ;
- `git checkout -b <nom>` pour créer une branche et changer la branche courante pour celle-ci.

Statut des branches

- La commande `git status` indique la branche courante ;
- La commande `git branch` montre la liste des branches et la branche courante.

Suppression d'une branche

- `git branch -d <nom>` pour supprimer une branche ;
- Si la branche n'a pas été fusionnée, `git branch -D <nom>`.

Application

```
$ git branch premiere_branche

$ git branch
* master
  premiere_branche

$ git checkout -b develop
Basculement sur la nouvelle branche 'develop'

$ git status
Sur la branche develop
rien à valider, la copie de travail est propre

$ git branch -d premiere_branche
Branche premiere_branche supprimée (précédemment 168efba)

$ touch dev.txt

$ git add -A && git commit -m ''Ajout fichier dev''
[develop 2292018] Ajout fichier dev
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dev.txt
```

Changer de branche

Plus facile qu'à l'UTC !

Comment faire ?

- Une commande simple : `git checkout <nom_de_la_branche>` ;
- **Attention :** ne pas avoir de changements non validés ! (ou passer par un `git stash`)

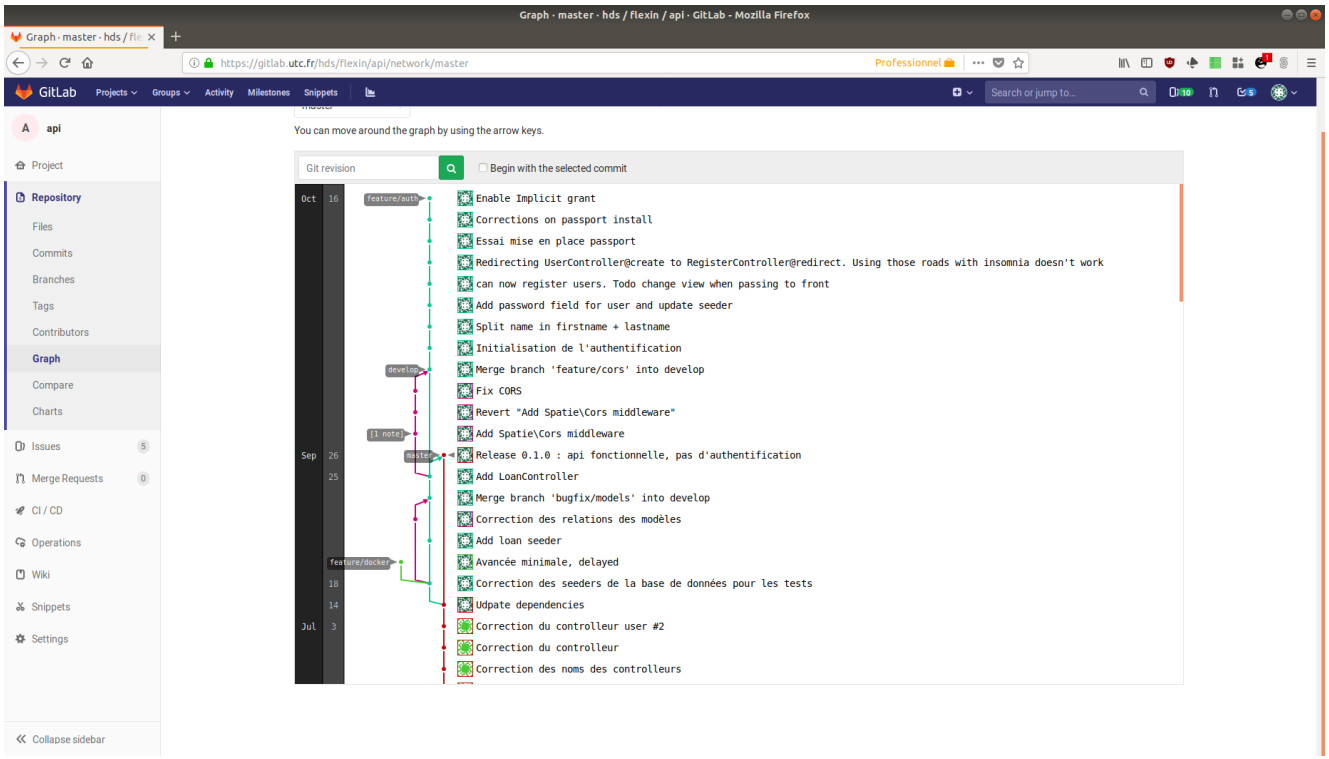
Rappel

`git checkout` sert aussi à se déplacer sur un commit précis.
En fait, cette commande sert à déplacer le `HEAD`. On peut assimiler un `checkout` sur une branche à un `checkout` sur un commit.
En fait, le nom de le branche n'est **qu'une étiquette** sur le dernier commit de celle-ci.

Visualisation

Un joli graphe !

- En console : `git log --graph --decorate --all`
- Depuis GitLab :



Visualisation

```
$ git checkout master && ls
Basculement sur la branche 'master'
Votre branche est à jour avec 'origin/master'.
API.txt

$ git log --graph --decorate --all
* commit 2292018a27182fef507601140c7f93d679b93678 (develop)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Mon Jan 14 14:01:37 2019 +0100
|
|     Ajout fichier dev
|
* commit 168efba77dcfd59ba4346fe4a34427b71db75da7 (HEAD -> master)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 14:53:09 2019 +0100
|
|     Commit supplémentaire
|
* commit a04da653083b6b0ba3eea2bce98d903acfd0a4d3
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 10:16:10 2019 +0100
|
|     Troisième commit
[...]
```

Fusionner deux branches avec Git

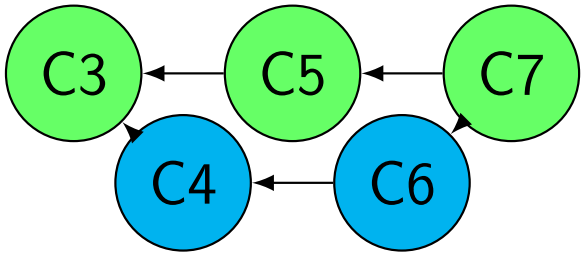
Attention : avant de commencer, s'assurer que le répertoire est dans un état propre et qu'il n'y a pas de modifications non validées !

Git propose deux façons de fusionner deux branches :

- le `merge` : fusion « basique » de deux branches
  C'est l'alternative la plus simple
- le `rebase` : modification plus avancée de l'historique
  Plus puissant mais plus dangereux

Le merge

Le `merge` applique les modifications apportées depuis la divergence par la branche secondaire sur la branche de départ.



Ici, C7 réalise le `merge` de la branche bleue dans la branche verte : les modifications apportées par C4 et C6 sont appliquées après C5.

Processus de fusion

1. se déplacer sur la branche destination de la fusion
   `git checkout <destination>`
2. lancer la fusion de la branche de travail
   `git merge <branche de travail>`

Le merge

Application

```
$ echo nouvelle ligne >> API.txt && git commit -am "Commit sur master"
[master 37d2273] Commit sur master
 1 file changed, 1 insertion(+)

$ git merge develop
Merge made by the 'recursive' strategy.
 dev.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dev.txt
```

Le merge

Log !

```
$ git log --graph --decorate

*   commit 1892470146ed37358a45e8e5cd9715ec0fc3448a (HEAD -> master)
|\  Merge: 37d2273 2292018
| | Author: huetremy <remy.huet@etu.utc.fr>
| | Date:   Mon Jan 14 14:50:00 2019 +0100
| |
| |     Merge branch 'develop'
| |
| * commit 2292018a27182fef507601140c7f93d679b93678 (develop)
| | Author: huetremy <remy.huet@etu.utc.fr>
| | Date:   Mon Jan 14 14:01:37 2019 +0100
| |
| |     Ajout fichier dev
| |
* | commit 37d2273def21ff6d8bb33f0600220764f67f2ab9
|/  Author: huetremy <remy.huet@etu.utc.fr>
|   Date:   Mon Jan 14 14:49:12 2019 +0100
|
|       Commit sur master
|
* commit 168efba77dcfd59ba4346fe4a34427b71db75da7
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Fri Jan 11 14:53:09 2019 +0100
|
|     Commit supplémentaire
```

Le rebase

Avertissement

ATTENTION – COMMANDE RISQUÉE

Le `rebase` cause une **modification de l'historique** qui peut rendre l'état de votre arbre incohérent avec une éventuelle branche distante (GitLab...).

Conclusion

On rebase une branche locale mais **JAMAIS** une branche synchronisée
(sauf si vous êtes absolument sûrs de ce que vous faites).

Le rebase

Comparaison avec le merge

Le `rebase` intègre aussi les modifications d'une branche secondaire sur la branche cible depuis la divergence mais ne crée pas de commit de merge.

Les commits effectués depuis la divergence sont **déplacés et réappliqués** après le dernier commit de la branche cible.

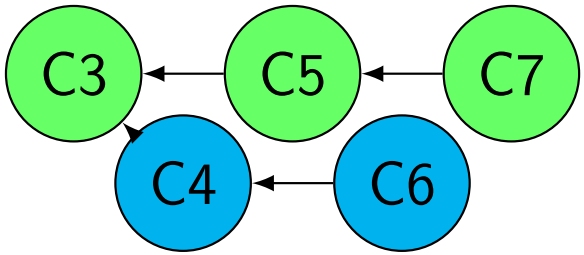


Figure – Avant le rebase

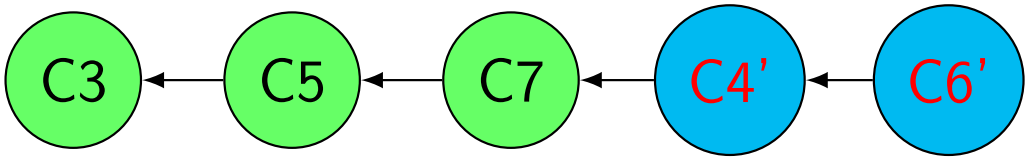


Figure – Après le rebase

Le rebase

Application simple

Après le rebase, pour ramener la branche de référence au bout de la chaîne de commits, on fait une fusion classique (merge).

Processus de rebase avec fusion

- se déplacer sur la branche à rebase
  `git checkout <branche à rebase>`
- rebase la branche courante sur la branche de référence
  `git rebase <branche de référence>`
- régler les éventuels conflits...
- se déplacer sur la branche de référence
  `git checkout <branche de référence>`
- fusionner la branche à rebase dans la branche de référence
  `git merge <branche à rebase>`

Le rebase

Exemple

```
$ echo "Avant rebase" >> API.txt && git commit -am "Avant rebase"
[master 6fe645e] Avant rebase
 1 file changed, 1 insertion(+)

$ git checkout develop && echo ''Ma première ligne'' >> dev.txt && git commit -am ''ajout à dev''
Basculement sur la branche 'develop'
[develop 59eed37] ajout à dev
 1 file changed, 1 insertion(+)
$ git log --graph --decorate --all
```

```
* commit 59eed37beb51329f95e90e6d6a6de85ca36c441a (HEAD -> develop)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Mon Jan 14 15:33:59 2019 +0100
|
|     ajout à dev
|
| * commit 6fe645e97ba1a69c99438fba05f21902d19f1055 (master)
| | Author: huetremy <remy.huet@etu.utc.fr>
| | Date:   Mon Jan 14 15:31:47 2019 +0100
| |
| |     Avant rebase
| |
| *   commit 1892470146ed37358a45e8e5cd9715ec0fc3448a
| |\  Merge: 37d2273 2292018
| | | Author: huetremy <remy.huet@etu.utc.fr>
|/| |  Date:   Mon Jan 14 14:50:00 2019 +0100
| | |
| | |       Merge branch 'develop'
| | |
* | | commit 2292018a27182fef507601140c7f93d679b93678
| | | Author: huetremy <remy.huet@etu.utc.fr>
| | | Date:   Mon Jan 14 14:01:37 2019 +0100
| | |
| | |     Ajout fichier dev
| | |
| * | commit 37d2273def21ff6d8bb33f0600220764f67f2ab9
|/  | Author: huetremy <remy.huet@etu.utc.fr>
|   | Date:   Mon Jan 14 14:49:12 2019 +0100
|   |
|   |       Commit sur master
|   |
* commit 168efba77dcfd59ba4346fe4a34427b71db75da7
[...]
```

```
$ git rebase master
Rembobinage préalable de head pour pouvoir rejouer votre travail par-dessus...
Application de  ajout à dev
$ git log --graph --decorate
* commit 1b9926f89453896c3f11e1f75b6871d70f35c5a4 (HEAD -> develop)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Mon Jan 14 15:33:59 2019 +0100
|
|     ajout à dev
|
* commit 6fe645e97ba1a69c99438fba05f21902d19f1055 (master)
| Author: huetremy <remy.huet@etu.utc.fr>
| Date:   Mon Jan 14 15:31:47 2019 +0100
|
|     Avant rebase
|
*   commit 1892470146ed37358a45e8e5cd9715ec0fc3448a
|\  Merge: 37d2273 2292018
| | Author: huetremy <remy.huet@etu.utc.fr>
| | Date:   Mon Jan 14 14:50:00 2019 +0100
| |
| |     Merge branch 'develop'
| |
| * commit 2292018a27182fef507601140c7f93d679b93678
| | Author: huetremy <remy.huet@etu.utc.fr>
| | Date:   Mon Jan 14 14:01:37 2019 +0100
| |
| |     Ajout fichier dev
| |
* | commit 37d2273def21ff6d8bb33f0600220764f67f2ab9
|/  Author: huetremy <remy.huet@etu.utc.fr>
|   Date:   Mon Jan 14 14:49:12 2019 +0100
[...]
```

```
$ git checkout master
$ git merge develop
Updating 6fe645e..1b9926f
Fast-forward
  dev.txt | 1 +
  1 file changed, 1 insertion(+)
$ git log --graph --decorate
```

On retrouve les mêmes log que sur `develop` : si vous regardez bien, il s'agit d'un merge de type *fast-forward*, donc `master` et `develop` pointent sur le même commit !

Le rebase

Rebase interactif – Principe

Cet outil puissant permet de ré-écrire comme bon vous semble votre historique **local**.

On ne le répètera jamais assez : **n'utilisez jamais cette commande sur des commits déjà poussés, sauf si vous savez exactement ce que vous faites et connaissez les conséquences.**

Cas d'utilisation

- Changer l'ordre des commits
- Fusionner des commits
- Supprimer des commits
- Modifier le contenu d'un ou plusieurs commits
- ...

Le rebase

Rebase interactif – Fonctionnement

Les commits que vous souhaitez modifier ou ré-ordonner vous sont présentés dans un éditeur. Pour chaque commit, vous pouvez choisir une action.

Principales commandes pour le rebase interactif

- `pick` : garder le commit en l'état
- `reword` : éditer le message de commit
- `edit` : modifier le contenu du commit
- `squash` : fusionner avec le commit précédent
- `drop` : supprimer le commit

Il est aussi possible de changer l'ordre des commits en les ordonnançant manuellement.

Le rebase

Rebase interactif – Exemple (1/4)

Vous avez développé une fonctionnalité, puis commencé une deuxième sans être certain de la garder en l'état, puis réparé la première.

```
$ echo "Ajout d'une super fonctionnalité" >> F1.txt
$ git add F1.txt && git commit -m "Nouvelle fonctionnalité : F1"
[master b191156] Nouvelle fonctionnalité : F1
1 file changed, 1 insertion(+)

$ echo "Début d'une autre fonctionnalité" >> F2.txt
$ git add F2.txt && git commit -m "wip F2"
[master dc4eba5] wip F2
1 file changed, 1 insertion(+)

$ echo "Rajout du point virgule manquant..." >> F1.txt
$ git commit -am "Résolution de bug sur F1"
[master a1b72f7] Résolution de bug sur F1
1 file changed, 1 insertion(+), 1 deletion(-)
```

Vous voulez maintenant pousser un historique propre pour faciliter la compréhension des autres contributeurs.

Le rebase

Rebase interactif – Exemple (2/4)

L'historique local est dans l'état suivant :

```
$ git log --oneline

a1b72f7  (HEAD -> master) Résolution de bug sur F1
dc4eba5 wip F2
b191156 Nouvelle fonctionnalité : F1
1b9926f (develop) ajout à dev
[...]
```

Objectifs

- **Fusionner** les commits concernant F1 (le bug est trop minime pour en faire un commit...)
- **Renommer** le commit concernant F2 (finalement, vous gardez votre développement en l'état).

Le rebase

Rebase interactif – Exemple (3/4)

On lance la commande de rebase, et on arrive dans un éditeur avec le contenu suivant.

Les commits les plus anciens sont en haut, contrairement aux logs

```
$ git rebase -i HEAD~3

pick b191156 Nouvelle fonctionnalité : F1
pick dc4eba5 wip F2
pick a1b72f7 Résolution de bug sur F1
```

On change l'ordre des commits ainsi que les commandes, puis on quitte l'éditeur.

```
pick b191156 Nouvelle fonctionnalité : F1
squash a1b72f7 Résolution de bug sur F1
reword dc4eba5 wip F2
```

Le rebase

Rebase interactif – Exemple (4/4)

Il nous faudra encore renseigner les nouveaux messages de commits pour la fusion et le renommage. On mettra un nom plus explicite à la place de « `wip F2` ». Une fois validés, Git joue les **nouveaux** commits un à un, comme pour un rebase classique.

```
[detached HEAD c76ff8d] Nouvelle fonctionnalité : F1
Date: Wed Jan 16 01:24:52 2019 +0100
1 file changed, 2 insertions(+)
create mode 100644 F1.txt
[detached HEAD db9f68a] Début de la fonctionnalité F2
1 file changed, 1 insertion(+)
create mode 100644 F2.txt
Successfully rebased and updated refs/heads/master.

$ git log --oneline

db9f68a (HEAD -> master) Début de la fonctionnalité F2
c76ff8d Nouvelle fonctionnalité : F1
1b9926f (develop) ajout à dev
[...]
```

Le rebase

Rebase interactif – Conclusion

Il y a beaucoup d'autres commandes : édition de commit, création de commits de fusion, exécution de commande shell... À adapter au besoin.

Soyez toujours prudents !

Par exemple, lorsque vous changez l'ordre de commits qui modifient le même fichier. Vous risquez de vous retrouver avec des **conflits en cascade** complexes à résoudre. En cas de problèmes, utilisez la commande `git rebase --abort` pour revenir en arrière, et ré-étudiez le problème.

Le cherry-pick

Ou comment récupérer des commits « à la demande »

Applique les changements introduits par un ou plusieurs commits.
Il faut éviter d'utiliser cette commande à tort et à travers, mais elle est très utile dans certains cas bien spécifiques.

Contrairement au rebase, on peut cherry-pick des commits déjà poussés.

Cas d'utilisation

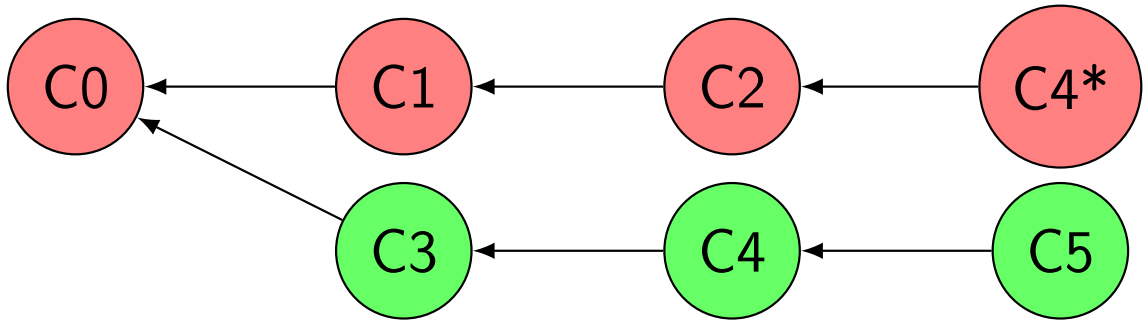
- Récupérer une fonctionnalité d'une branche sans le reste
- Rétropropager la correction d'un bug faite sur une nouvelle branche
- ...

Le cherry-pick

Exemple (1/2)

Je règle un bug sur une branche `develop`, et je continue à développer. Je veux que la résolution soit portée sur `master`, mais sans récupérer les fonctionnalités instables de `develop`.

On verra plus tard comment éviter cette situation pas très propre. . . .



Comment récupérer uniquement `C4` ?

Le cherry-pick

Exemple (2/2)

```
$ git checkout develop
$ cat dev.txt
Ma première ligne
$ echo "Resolution de bug" >> dev.txt
$ git commit -am "Resolution de bug"
[develop c586d4d] Resolution de bug
  1 file changed, 1 insertion(+)
$ echo "Debut nouvelle fonctionnalité" >> dev.txt
$ git commit -am "Nouvelle fonctionnalité"
[develop 2bbe0bd] Nouvelle fonctionnalité
  1 file changed, 1 insertion(+)
$ cat dev.txt
        Ma première ligne
        Resolution de bug
        Debut nouvelle fonctionnalité
$ git checkout master
Switched to branch 'master'
$ git show HEAD --oneline
db9f68a (HEAD -> master) Début de la fonctionnalité F2
$ git cherry-pick -x c586d4d
[master 6bc2b57] Resolution de bug
  1 file changed, 1 insertion(+)
$ git show HEAD
6bc2b57 (HEAD -> master) Resolution de bug (cherry picked from commit c586d4d)
$ cat dev.txt
        Ma première ligne
        Resolution de bug
```

Le cherry-pick

Conclusion

Recommandations

- Préférer le merge au cherry-pick quand c'est possible
- Le cherry-pick n'évite pas les conflits, par exemple si un fichier a été modifié sur les deux branches depuis le commit ancêtre commun
- Toujours utiliser le flag `-x` pour retrouver le commit de référence
- Utiliser le flag `-e` pour modifier le message de commit

Organiser tout ça

Il existe de nombreuses méthodes et modèles pour organiser tout ce joyeux bazar à base de branches, d'issues et de merge request. L'une d'entre elles est git flow.
Git flow est un workflow, un modèle définissant où stocker quelle information et dans quel ordre exécuter quelles tâches ; mais c'est aussi un outil en ligne de commande qui permet d'accélérer la mise en place de ce workflow.
Il faut l'installer en plus de git. Sous debian et dérivées :

```
# apt update && apt -y install git-flow
```

Le workflow

Problématique de collaboration sur des gros projets :

- Avoir une branche `master` stable à tout moment ;
- Créer une branche par nouvelle fonctionnalité :
  - Pour une organisation plus claire ;
  - Pour développer plusieurs fonctionnalités en parallèle.
- Avoir une branche de développement pour merge les fonctionnalités finies, mais qui n'est pas une branche stable.

Comment ça fonctionne ?

Les branches master et develop

La branche master

Il s'agit d'une branche stable. À tout moment son commit le plus récent correspond à une version fonctionnelle du projet.

La branche develop

C'est la branche de travail courante. C'est sur celle-ci qu'on ajoute au fur et à mesure les fonctionnalités, et que l'on mergera dans master pour effectuer une release

Comment ça fonctionne ?

Feature, bugfix, release et hotfix

Ce sont des branches avec des utilités bien définies :

feature

Une fonctionnalité particulière, qui sera merge dans develop quand elle sera finie

bugfix

Résolution d'un bug **qui n'existe que sur develop**. Sera merge dans develop

release

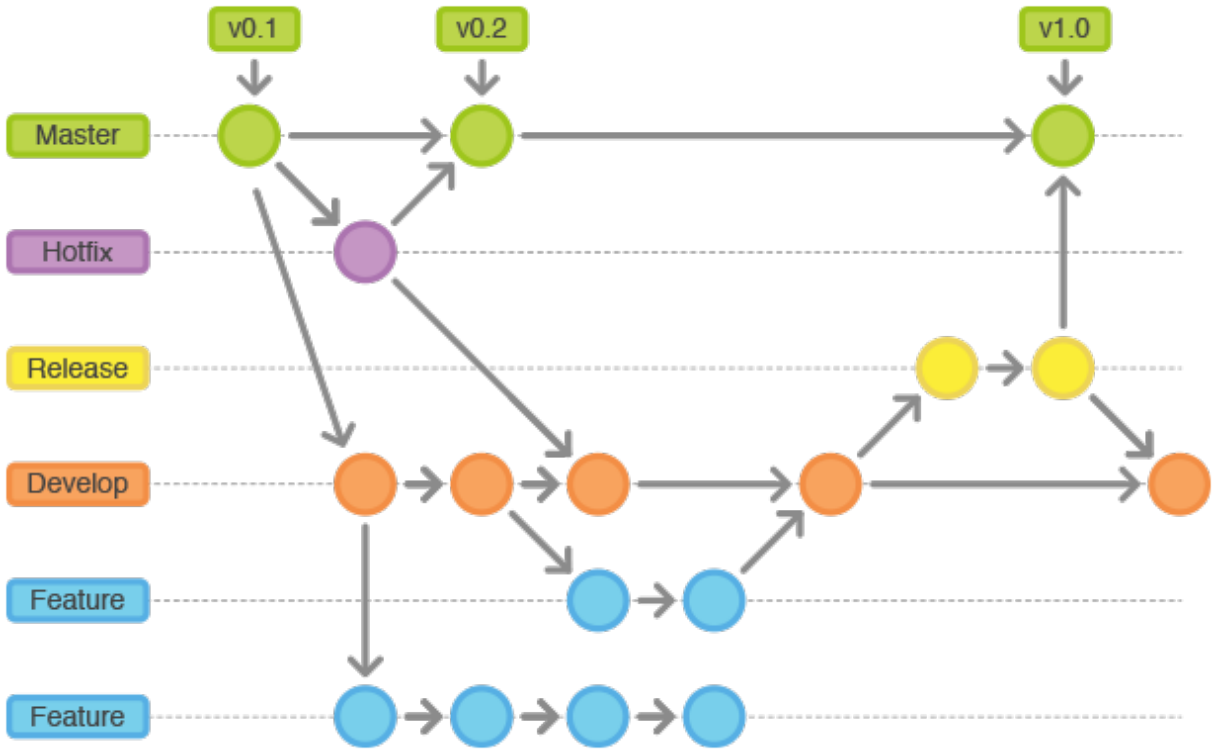
Permet de faire des modifications sur le projet avant la sortie d'une release. Merge dans master et dans develop (crée un tag sur master)

Comment ça fonctionne ?

Hotfix

hotfix

Pour résoudre un bug qui est présent sur une version « stable » (donc dans master). Sera merge dans develop **et** dans master (crée un tag sur master).



En pratique

init

Tout commence par un **init**, à l'intérieur d'un repo. Appuyez sur entrée à chaque question, on reste sur les noms par défaut.

```
git flow init
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [./.git/hooks]
```

En pratique

Si vous êtes développeur : créer des features

Pour créer une feature du nom de pigeon :

```
git flow feature start pigeon
Basculement sur la nouvelle branche 'feature/pigeon'

Summary of actions:
- A new branch 'feature/pigeon' was created, based on 'develop'
- You are now on branch 'feature/pigeon'

Now, start committing on your feature. When done, use:

     git flow feature finish pigeon
```

Git flow a créé la branche feature/pigeon.

En pratique

Si vous êtes peu de développeurs : merge des features

```
git flow feature finish pigeon
Basculement sur la branche 'develop'
Déjà à jour.
Branche feature/pigeon supprimée (précédemment 83fb6b9).

Summary of actions:
- The feature branch 'feature/pigeon' was merged into 'develop'
- Feature branch 'feature/pigeon' has been locally deleted
- You are now on branch 'develop'
```

Git flow a merge la feature dans develop et a supprimé la branche feature/pigeon.

En pratique

Si vous êtes une grande équipe : push des features

```
git flow feature publish pigeon
```

Ceci va tout simplement push la branche feature/pigeon sur le serveur. On pourra éventuellement faire une merge request pour la mettre dans develop. Pour pull les features des autres :

```
git flow feature pull origin pigeon
```

En pratique

La release : pour les changements de dernière minute

```
git flow release start pigeon2000
Basculement sur la nouvelle branche 'release/pigeon2000'

Summary of actions:
- A new branch 'release/pigeon2000' was created, based on 'develop'
- You are now on branch 'release/pigeon2000'

Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:

     git flow release finish 'pigeon2000'
```

Résumé des comandes