



FORMATION GIT

OBJECTIFS

- Le but de cette formation est de préparer les participants au monde de Git
- Pour commencer, les personnes connaissant SVN doivent oublier tout ce qu'elles savent
- Il en est de même pour les utilisateurs de CVS
- Maintenant que c'est fait, le cours va porter sur les points suivants
 - Qu'est-ce que le versionning (CVCS/DVCS)
 - Pourquoi est-ce que Git est une bonne idée
 - Comment travailler en local avec Git
 - Comment travailler en équipe
 - Configuration et outils pour Git





PLAN

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes
- Mettre en place des organisations de travail
- Conclusion
- Projets OpenSource & Pull Requests
- Allons plus loin



LOGISTIQUE

- Horaires
- Déjeuner & pauses
- Autres questions ?





INTRODUCTION



PLAN

- *Introduction*
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes
- Mettre en place des organisations de travail
- Conclusion
- Projets OpenSource & Pull Requests
- Allons plus loin





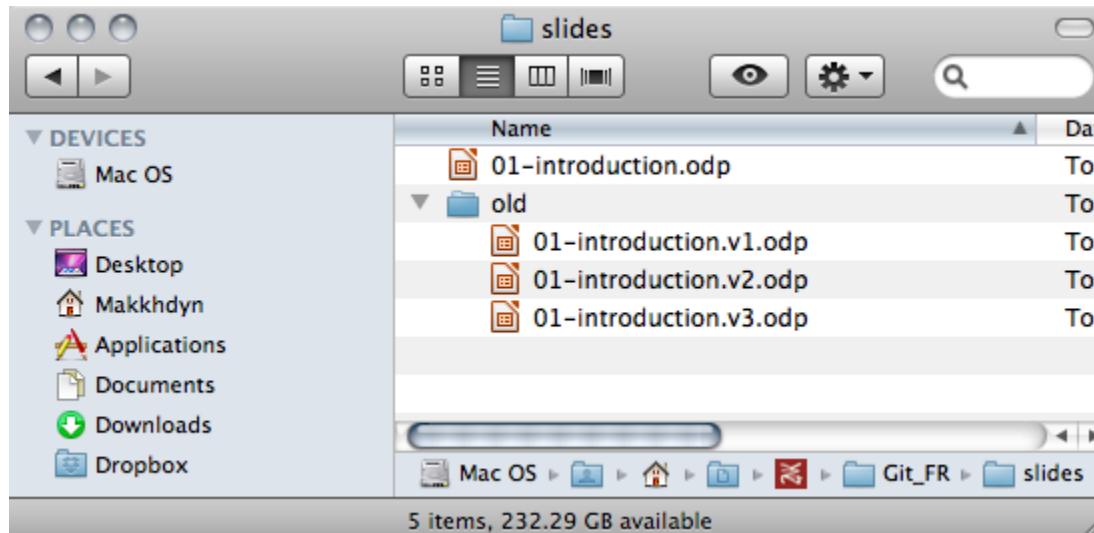
BESOIN ?

- Sauvegarde d'un contenu de façon à sécuriser les derniers ajouts/modifications
- Historisation des sauvegardes pour éviter la perte d'informations au cours du temps
- Cas standards d'utilisation
 - Développement d'une application
 - Écriture d'un rapport
 - Bases de données
- Simplement toute activité résultant de la création ou suppression de données peut bénéficier de versioning



LES MÉTHODES DE BASE

- Besoin de sauvegarde important en informatique depuis toujours
 - Débute souvent avec un dossier rempli de fichiers copiés



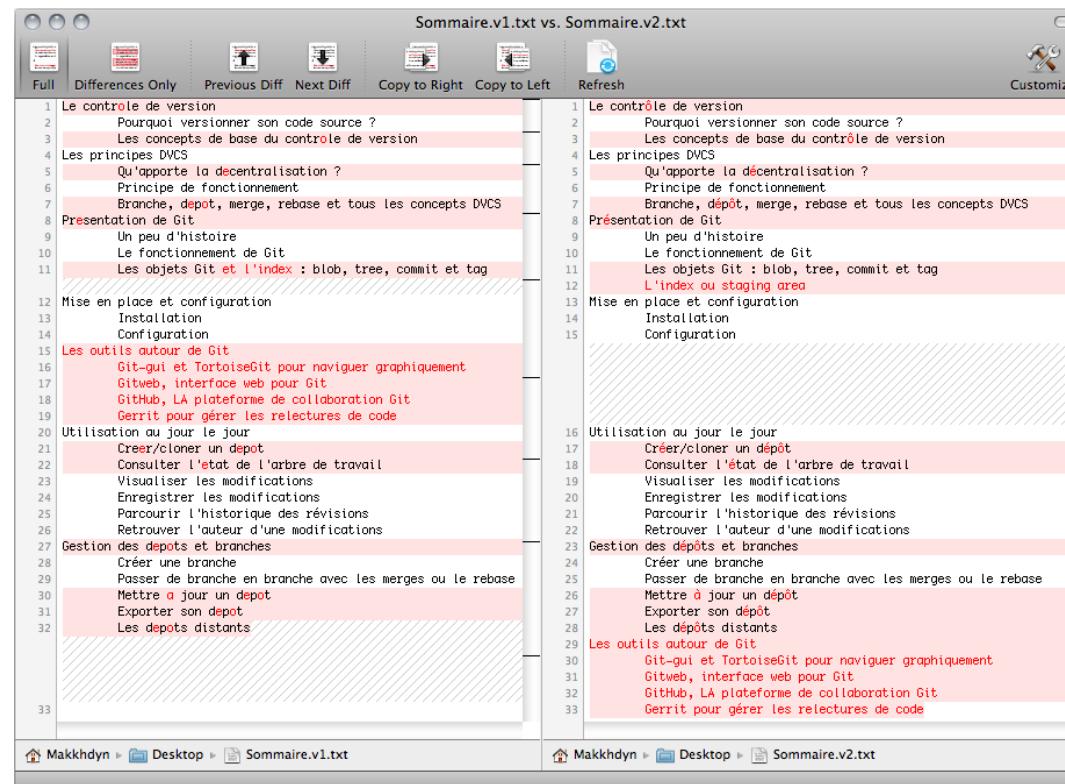
© Copyright 2021 Zenika. All rights reserved

Bien plus améliorée avec des systèmes plus avancés



LES AVANTAGES

- Conserver les données permet de voir l'évolution du contenu et les modifications apportées entre deux versions



- Ou simplement de revenir en arrière en cas d'erreur



DES SOLUTIONS COLLABORATIVES

- Seulement il est courant de ne pas être seul sur un projet et d'avoir besoin d'échanger les versions de chacun puis les fusionner



- Pour ça des systèmes de versioning plus élaborés ont été adoptés
- Regroupés sous le terme de V.C.S pour **Version Control System**
- Plusieurs fonctionnements existent selon les solutions choisies
 - Le fonctionnement traditionnel **centralisé** ou CVCS
 - Le fonctionnement **distribué** ou DVCS





LA NOTION DE "COMMIT"

- Quel que soit le type de versioning choisi, le fonctionnement est basé sur les **commits**
- Un commit est un regroupement de modifications (ajout/suppressions) auquel sont associés - a minima - un message, un auteur, une date
- Un commit doit se suffire à lui même :
 - Il doit laisser l'ensemble du contenu dans un état cohérent
 - Le message associé doit indiquer le contenu des modifications
 - Il est trop souvent négligé





CENTRALIZED VERSION CONTROL SYSTEM



CVCS - LES SOLUTIONS

- Le principe de CVCS a connu une très forte popularité grâce à deux solutions open-source
 - CVS (**C**oncurrent **V**ersions **S**ystem)
 - Projet de la FSF depuis 1990, créé par Dick Grune en 1986 et basé (à l'origine) sur des scripts shell
 - La dernière mise à jour date de mai 2008
 - SVN (**S**ub**v**ersion**n**)The logo for Subversion consists of a stylized blue 'S' icon followed by the word 'SUBVERSION' in a bold, sans-serif font. The 'S' has three horizontal bars above it and a series of dots below it. A horizontal blue line runs behind the 'S' and across the word 'SUBVERSION'.
 - Projet de l'ASF depuis 2009, créé en 2000 par l'entreprise CollabNet dans la vue de faire un successeur amélioré de CVS



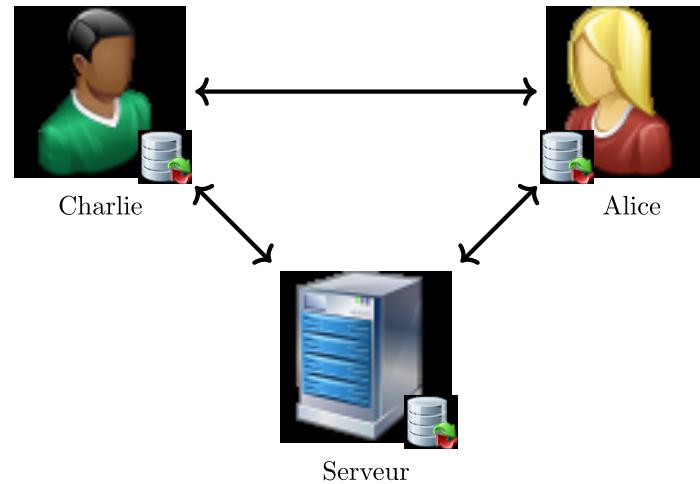
LES DÉFAUTS DU CVCS

- L'idée d'un serveur central possède quelques défauts
 - Impossible de versionner sans tout rendre public
 - Tout le monde a le droit de commit (et peut tout casser)
 - Tendance à l'unique commit massif
 - Chaque tâche requiert une connexion vers le serveur
- Les CVCS sont pratiques, mais dans certains cas, simplement inadaptés
 - Un serveur central implique une confiance totale envers ce serveur
 - Dans une grande organisation, on se marche sur les pieds lors de commits



DISTRIBUTED VERSION CONTROL SYSTEM

- Résolution des défauts de centralisation par une autre approche



- Chacun a son dépôt, et peut échanger avec tout le monde
- Un ou plusieurs dépôts servent à publier "publiquement" le contenu
- Chaque utilisateur est totalement autonome et peut faire ce qu'il souhaite dans son dépôt



DVCS - LES SOLUTIONS

- L'idée de faire un réseau de pair à pair n'est pas neuve, mais ce sont en particulier trois solutions (open-source) qui sont à l'origine de la notoriété des DVCS
 - Bzr (Bazaar) 
 - Crée par Martin Pool en février 2005 pour Canonical. Version 0.0.1 sortie le 26 mars 2005
 - Hg (Mercurial) 
 - Crée par Matt Mackal le 19 avril 2005, en réponse à l'arrêt de l'offre gratuite de BitKeeper (un DVCS propriétaire)
 - git (Git) 
 - Crée par Linus Torvalds le 7 avril 2005, en réponse à l'arrêt de l'offre gratuite de BitKeeper

*Mercurial does that too, but Git does it better... -
Linus Torvalds*



AVEC QUI ÉCHANGER ?

- Le système centralisé donne l'impression qu'il faut un seul dépôt
 - C'est faux, il suffit d'avoir des dépôts de **confiance**
 - Il n'est pas obligatoire de faire confiance à tout le monde
 - Il n'est pas non plus obligatoire de ne faire confiance qu'à un seul dépôt





EN BREF

- Les DVCS ont donc l'avantage d'être indépendants
 - on peut versionner du travail pas tout à fait fini
 - il est facile de faire un "fork" et de créer son propre projet (pratique dans l'open-source)
 - le travail est fait en local, pas besoin de connexion réseau pour toutes les opérations
 - chaque dépôt contient toutes les informations, il est impossible de perdre des données propagées
 - l'échange des modifications suit un workflow adapté/adaptable
- Et aucune de ces capacités n'est spécifique à Git !



QUESTIONS





FONCTIONNEMENT DE GIT



PLAN

- Introduction
- *Fonctionnement de Git*
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes
- Mettre en place des organisations de travail
- Conclusion
- Projets OpenSource & Pull Requests
- Allons plus loin



HISTORIQUE

- Le kernel Linux était versionné sur le DVCS propriétaire BitKeeper
 - Ce dernier annonce la fin de son offre gratuite suite à une polémique
 - Aucune alternative gratuite ne convient
- Le 3 avril 2005, Linus Torvalds débute le développement de Git
- Le 7 avril, Git est versionné sous Git
- Le 16 juin, le kernel linux est versionné sous Git
- Le 26 juillet, Junio Hamano devient le responsable du projet



INSTALLATION & QUI SUIS-JE ?

- Git est
 - un logiciel de DVCS open-source, multi-plateforme et gratuit
 - disponible sur <http://git-scm.com/>
 - basé sur les principes vus jusqu'à présent



- Après installation, un nom d'utilisateur et un email de contact doivent être configurés (et seront associés aux commits)

```
$ git config --global user.name "Zenika"  
$ git config --global user.email "training@zenika.com"
```



TRAVAILLER AVEC SES OUTILS

- Pour la gestion de certains messages et éditions, Git va automatiquement appeler un éditeur de texte
- Par défaut celui pointé par la variable d'environnement **EDITOR** sera choisi
- Pour spécifier un autre éditeur

```
| git config --global core.editor "notepad.exe"
```
- Certains éditeurs sont conseillés
 - Windows : notepad.exe notepad++.exe
 - Linux/UNIX : vim, emacs, nano
 - Mac OS : textmate



INITIALISER UN PROJET

- Une fois que Git est installé et configuré il ne reste plus qu'à démarrer un projet
- En ligne de commande, `git init` créera un dossier versionné sous Git

```
$ git init resaroute
Initialized empty Git repository in /home/user/resaroute/.git/
```

- Le dossier `.git` créé sera le lieu où toutes les données et configurations seront stockées en interne
 - Il n'y a qu'un seul et unique dossier `.git` pour tout le projet
 - Au fur et à mesure de la formation, le contenu de ce dossier sera analysé en détail



TROUVER DE L'AIDE

- En cas de problème, il est possible de se référer à la documentation Git via `git help <command>` ou `man git-command`
- La documentation est aussi disponible <http://git-scm.com/docs>

git(1) Manual Page

NAME

git - the stupid content tracker

SYNOPSIS

```
git [--version] [--help] [-c <name>=<value>]
```

- La mailing list git@vger.kernel.org permet de résoudre la plupart des problèmes techniques et bugs liés à Git
- La page <http://git-scm.com/documentation> liste les ressources les plus pertinentes dans le domaine



EXPLOITER L'AIDE 1/2

git-commit(1) Manual Page

NAME

git-commit - Record changes to the repository

SYNOPSIS

```
git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
           [--dry-run] [(-c | -C | --fixup | --squash) <commit>]
           [-F <file> | -m <msg>] [--reset-author] [--allow-empty]
           [--allow-empty-message] [--no-verify] [-e] [--author=<author>]
           [--date=<date>] [--cleanup=<mode>] [--[no-]status]
           [-i | -o] [-S[<keyid>]] [--] [<file>...]
```

DESCRIPTION

Stores the current contents of the index in a new commit along with a log message from the user.



EXPLOITER L'AIDE 2/2

- Une section avec le nom et une description courte
- La liste des options : []=**optionnel** , |=**ou** ,
- Une description plus complète
- Le détail des paramètres et les valeurs possibles



GIT DE L'INTÉRIEUR

- La structure interne de Git est un graphe, toutes les commandes modifient le graphe.
- Regardons un peu la structure interne de Git : les objets **blob**, **tree** et **commit**.



LE PROBLÈME À RÉSOUDRE

- Pour rappel : le système de versionning a pour but de sauvegarder et d'historiser du **contenu**
 - Pas des fichiers, ni une arborescence, mais bien du contenu
 - Chaque version différente = un contenu différent
- L'idée est maintenant de pouvoir identifier un contenu par rapport à un autre
 - La solution est de lui donner un identifiant unique
 - Pour deux contenus identiques un seul identifiant est généré

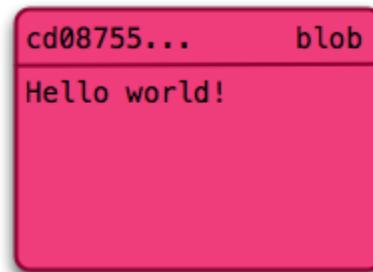


- Cette empreinte unique correspond à un Hash
 - SHA1 sera utilisé comme fonction de Hash



BLOB

- Le contenu de chaque fichier est pris, hashé et stocké dans une base interne
- La représentation la plus simple est celle-ci



```
$ git cat-file -p cd08755  
Hello world!
```

- Seul le **contenu** est enregistré, le nom et l'emplacement du fichier ne sont pas disponibles



TREE

- Puisque seul le contenu est conservé il faut un système pour réassigner celui-ci à un fichier. Cela se fera avec un **tree**

```
b7868f7...          tree
0cae752...  tree  firstDir
cd08755...  blob  firstFile.txt
```

```
$ tree
|- firstDir
|  |- firstFileCopy.txt
|  '- secondFile.txt
`- firstFile.txt
$ git cat-file -p b7868f7
040000 tree 0cae752... firstDir
100644 blob cd08755... firstFile.txt
```



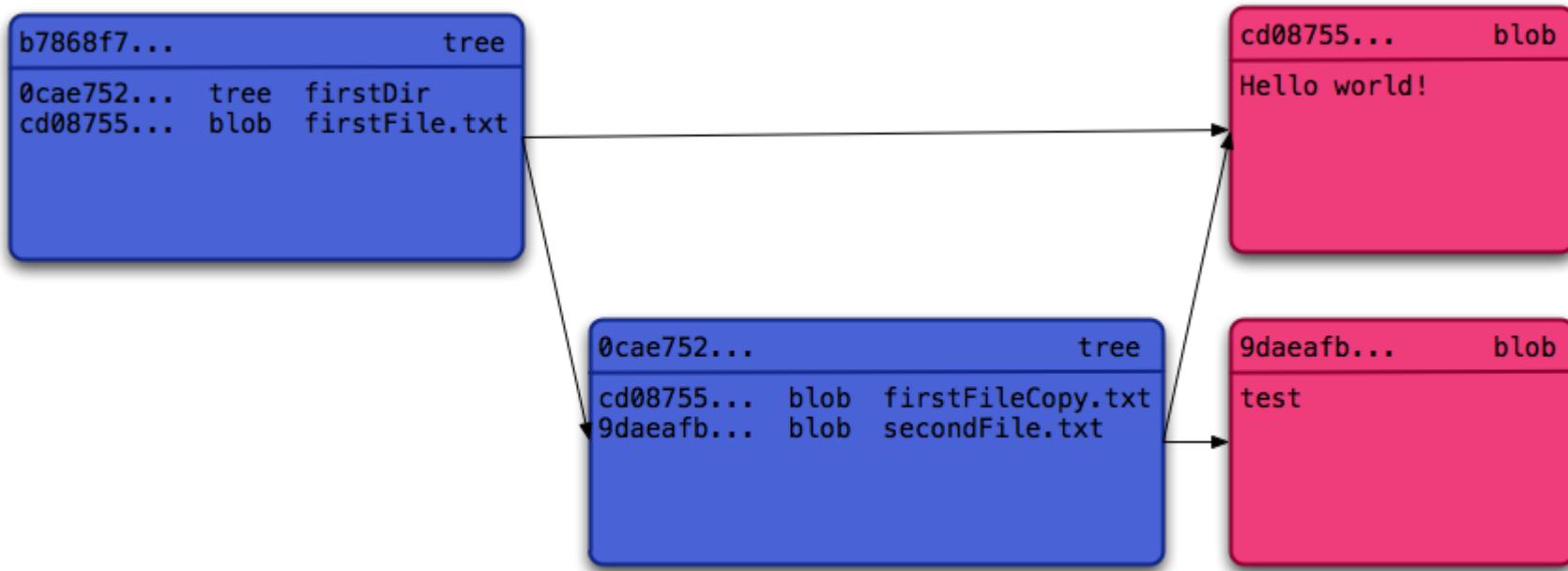


TREE

- Un tree contient des références vers des blobs ainsi que d'autres trees pour créer un système hiérarchique
- Cette fois-ci seuls les noms des "fichiers/dossiers" sont enregistrés
- Le SHA1 associé est celui qui permet d'accéder au blob/tree correspondant
- PS : Il ne peut pas exister de "dossier" (tree) vide !



EXEMPLE D'ARBORESCENCE



- Note : Il n'y a qu'un seul blob contenant "Hello world !"
- Même contenu => même SHA1



COMMIT

- Il ne manque plus qu'à obtenir les informations différenciant les versions des données. Les **commits**

```
64bf0dd...          commit
tree  b7868f7...
author Colin Hebert <..>
committer Colin Hebert <..>

First commit
```

```
$ git cat-file -p 64bf0dd
Tree b7868f7...
Author Zenika <training@zenika.com> 1308746088 +0100
Committer Zenika <training@zenika.com> 1308746088 +0100

First commit
```

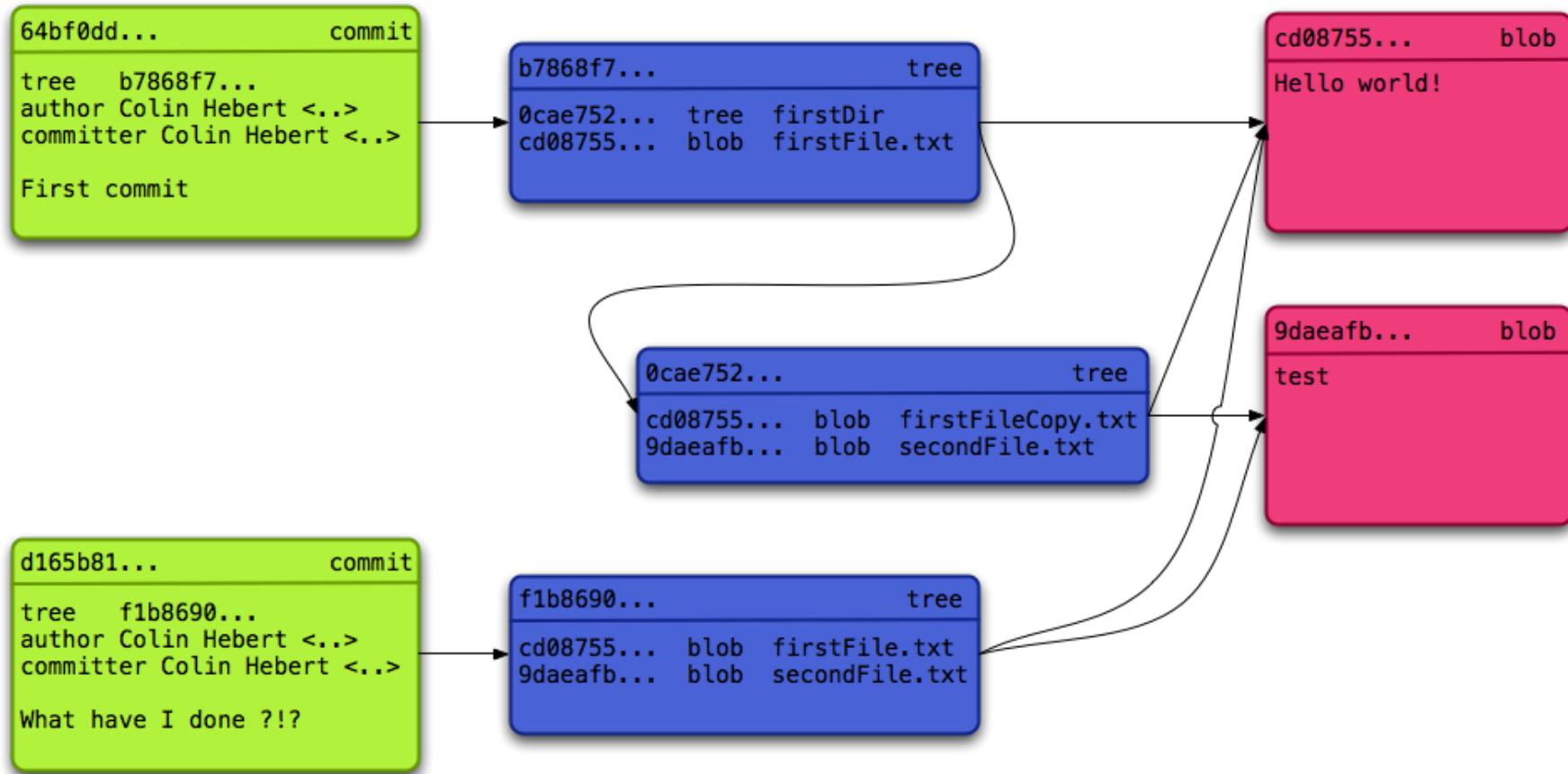


COMMIT

- Un commit contenant les méta-données d'une sauvegarde
 - Auteur, Timestamp, Tree servant de racine, message de commit
- Il y a une différence entre l'auteur et le commiteur (utile en OSS)
 - Auteur : a écrit le patch/la version actuelle
 - Commiteur : a validé le patch et avait les droits suffisants pour le mettre sur le dépôt



PLEIN DE COMMITS

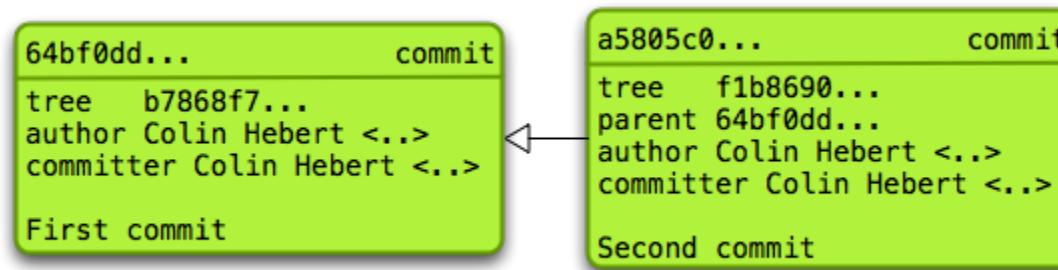


- Une petite explication de ce graphe ?



L'OEUF ET LA POULE

- En ayant plusieurs commits, il reste encore le problème de l'**ordre** des commits
- Pour ça un attribut **parent** est placé dans chaque commit, pointant sur le(s) commit(s) précédent(s)



- Contrairement à d'autres VCS l'ordre de parution des commits n'influe pas sur l'ordre réel des commits
- Chaque commit sait d'où il "vient", mais ne sait pas vers où il "va"
 - Il pourrait y avoir plusieurs branches (ou aucune)
 - Quand le commit est fait, il n'y a pas de commit suivant !



QUELQUES COMMANDES DE BASES

- `git add` : permet de dire à Git que l'on a modifié un fichier
- `git commit` : permet de sauvegarder l'état dans Git
- `git cat-file [-p | -t] <sha1>` : permet d'afficher le type (-t) ou le contenu (-p) d'un objet Git

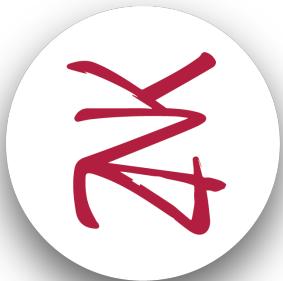


QUESTIONS





Lab 1



UTILISER GIT EN LOCAL



PLAN

- Introduction
- Fonctionnement de Git
- *Utiliser Git en local*
- Les références
- Utiliser Git en distant
- Configuration et outils externes
- Les workflows de travail
- Conclusion
- Projets OpenSource & Pull Requests
- Allons plus loin



ÉTAPES LORS D'UNE SAUVEGARDE

- Le système de sauvegarde de version par Git repose sur le maintien d'un objet "tree" temporaire et la validation de celui-ci lors de la création du commit
- Cet "arbre temporaire" est appelé l'index, la staging area ou encore le cache et contient toutes les informations en préparation d'un commit

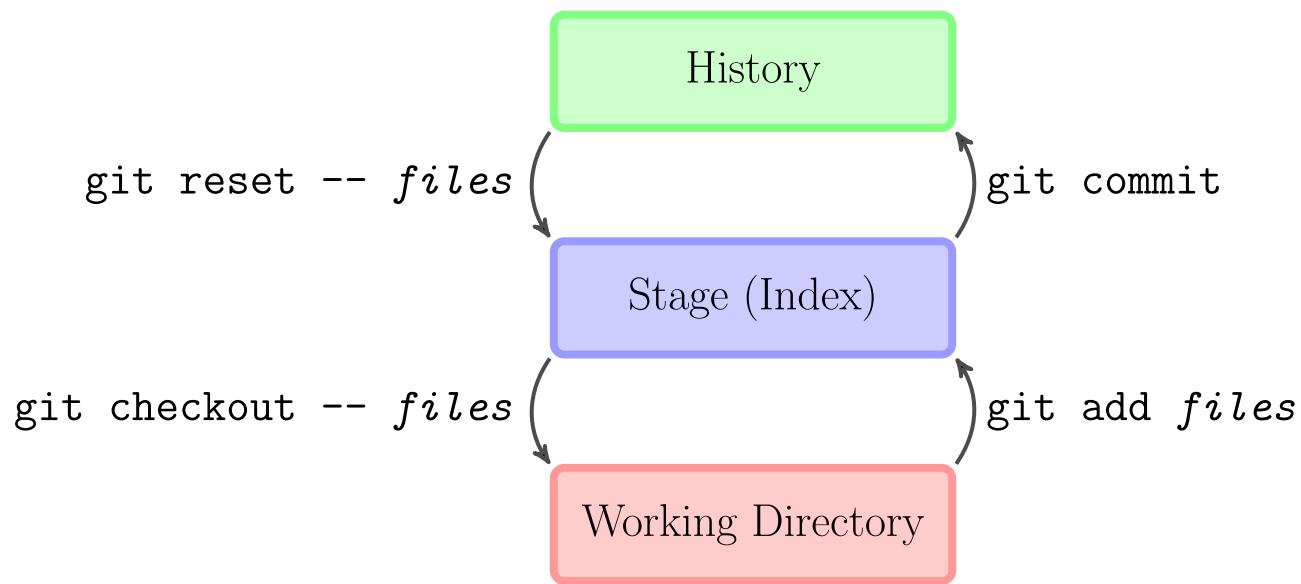


- Étapes de création du commit
 - Création/Modification/Suppression de contenu
 - Insertion des changements dans l'index
 - Validation de l'index dans l'état actuel et création de l'objet commit



WORKING/INDEX/STORED

- Les trois différentes zones où se situe le contenu sont donc
 - Working directory
 - Index (stage)
 - Base interne de Git



ÉTAT DE L'INDEX

- `git status` permet d'observer l'état actuel de l'index
 - Les éléments ajoutés/supprimés/modifiés
 - Les éléments non versionnés
- Cette commande permet d'observer rapidement toutes les actions qui ont été réalisées et qui n'ont pas encore été commitées

```
$ git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
$ echo "test" > newFile.txt
$ git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    newFile.txt
nothing added to commit but untracked files present (use "git add" to track)
```



AJOUTER/MODIFIER DES ÉLÉMENTS

- Ajouter ou modifier un élément signifie
 - Mettre le nouveau contenu dans la "base interne" de Git (blob)
 - Modifier l'index afin d'avoir un tree à jour pour un futur commit
- Afin de simplifier ces deux opérations, une seule commande suffit, **git add**

```
git add [-A] [-i] [-e] [<file-pattern ...>]
```

-A Ajoute tous les éléments non-versionnés et modifiés à l'index
-i Lance l'ajout des éléments en mode interactif
-e Ouvre un éditeur pour modifier l'**élément avant qu'il** ne soit versionné
<file-pattern ...>
Fichiers à prendre en compte lors de l'ajout à l'index
Les Fileglobs sont autorisés

- Après ajout via **git add**, les nouveaux blobs ont été ajoutés en base, et le tree est prêt à être commité ou remodifié



GIT ADD

- D'abord les contenus sont modifiés/créés

```
$ echo "test" >> newFile.txt; echo "test" >> firstFile.txt
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
      modified: firstFile.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    newFile.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

- Un simple appel à `git add` ajoute toutes les informations dans l'index

```
$ git add -A
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged HEAD <file>..." to unstage)
    modified: firstFile.txt
    new file: newFile.txt
```



GIT ADD EN MODE INTERACTIF

- Une autre façon de faire consiste à passer par le mode interactif

```
$ git add -i
      staged          unstaged path
  1: unchanged          +1/-0 firstFile.txt

*** Commands ***
1: status   2: update    3: revert    4: add untracked
5: patch   6: diff       7: quit      8: help
What now>
```

- Le mode interactif permet une utilisation avancée

- Ajout partiel de contenu
- Génération de patchs
- Annulation de modifications
- Détail des modifications effectuées

- Le tout avec une application en ligne de commande détaillée



SUPPRESSION DE CONTENU

- La suppression de contenu permet de retirer au fur et à mesure le contenu versionné devenu inutile
 - Supprimer un contenu ne supprime pas les anciennes versions de celui-ci, mais l'enlève uniquement pour les prochaines versions
- La commande de suppression est **git rm**, elle se charge aussi de supprimer le fichier du disque dur si celui-ci est encore présent

```
git rm [-r] [--cached] [<file-pattern ...>]
```

-r Supprime les dossiers (trees) récursivement

--cached Supprime le contenu de l'index uniquement
et ne touche pas aux fichiers/dossiers

<file-pattern ...>

Fichiers à prendre en compte lors de la suppression de l'index

Les Fileglob sont autorisés



GIT RM

- La suppression est très simple, les fichiers déjà supprimés sont signalés dans le `git status`

```
$ rm secondFile.txt
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged HEAD <file>..." to unstage)

    modified: firstFile.txt
    new file: newFile.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

    deleted: secondFile.txt
```



GIT RM

- La suppression de l'index et du fichier si nécessaire

```
$ git rm secondFile.txt
rm 'secondFile.txt'
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged HEAD <file>..." to unstage)

    modified: firstFile.txt
    new file: newFile.txt
    deleted: secondFile.txt
```



VISUALISATION DES MODIFICATIONS

- `git diff` permet d'afficher les modifications au format textuel diff unifié

```
git diff [--cached] [<commit1> [<commit2>]] [<file-pattern ...>]
```

`--cached`

Permet de comparer l'index par rapport au dépôt

`<commit1>`

Commit à partir duquel comparer

`<commit2>`

Commit avec lequel comparer (par défaut le commit courant)

`<file-pattern ...>`

Fichiers à comparer

Les Fileglob sont autorisés

- Par défaut Git compare le contenu du working directory avec celui de l'index



GIT DIFF

- Les modifications entre l'index et le dépôt

```
$ git diff --cached
diff --git a/firstFile.txt b/firstFile.txt
index 980a0d5..36310c8 100644
--- a/firstFile.txt
+++ b/firstFile.txt
@@ -1 +1 @@
-Hello World!
+Hello people!
```

- Les modifications entre le working directory et l'index

```
$ git diff
diff --git a/firstFile.txt b/firstFile.txt
index 36310c8..e26f92a 100644
--- a/firstFile.txt
+++ b/firstFile.txt
@@ -1 +1,2 @@
  Hello people!
+Goodbye people!
```



VALIDATION DE L'INDEX ET COMMIT

- Une fois l'index modifié et acceptable pour une sauvegarde, il ne reste plus qu'à l'enregistrer grâce à un commit
- La commande `git commit` s'occupe de cette étape

```
git commit [-a|--interactive] [-v] [--amend] [-m <msg>]
```

`-a` Ajoute tous les contenus modifiés/supprimés à l'index avant le commit

`--interactive`

Lance le mode interactif de `git add` avant le commit

`-v` Mode verbeux, affiche un diff avant de valider le commit

`--amend`

Supprime le commit précédent et le remplace par le courant

Le message de commit sera initialisé au message précédent

Pratique lorsque l'`on s'est trompé` lors d'un commit

`-m <msg>`

Message à utiliser lors du commit. Si aucun message n'est fourni

Git essayera d'ouvrir un éditeur de texte pour recevoir un message de commit

Outre le fait qu'il soit recommandé, par défaut il n'est pas possible de commiter sans message



GIT COMMIT

- L'appel de `git commit` avec l'option `-m` permet de tout faire en une seule ligne de commande

```
$ git commit -m "A relevant message"  
[master 7cc6ab3] A relevant message  
 3 files changed, 2 insertions(+), 1 deletions(-)  
  create mode 100644 newFile.txt  
  delete mode 100644 secondFile.txt  
$ git status  
On branch master  
nothing to commit, working directory clean
```

- Bien sûr, après un commit, l'index est complètement neuf et correspond exactement au contenu du dernier commit
- Attention : En équipe ou seul, avoir des messages de commit pertinents est important pour potentiellement revenir sur d'anciennes sauvegardes



DES COMMITS ET DES MESSAGES

- Au fur et à mesure, une norme s'est installée au sein des messages
- Celle-ci veut que chaque message respecte le format suivant
 - Titre : Simple concis et en moins de 50 caractères et écrit au présent
 - Contenu : Un texte plus long contenant un détail des opérations effectuées et leurs raisons. Dans le cas de l'utilisation d'un bug tracker le numéro du bug corrigé
- Les deux points étant séparés par une ligne vide
- Ce format vient du format des mails, avec objet et contenu
- <http://bit.ly/goodcommitmessages>



VOIR L'HISTORIQUE

- La commande `git log` permet de lister, chercher et formater les commits selon certains critères

```
git log [-<n>] [--pretty[=<format>]] [--abbrev-commit] [--oneline]  
[--graph] [--all|<since>..<until>] [<path>...]
```

`-<n>`

Liste n commits uniquement

`--pretty[=<format>]`

Formate la sortie sous les formes oneline, short, full, fuller, medium (par défaut), email, raw ou format:`<string>`

`--abbrev-commit`

Affiche les identifiants de commits en raccourci

`--oneline`

Raccourci pour "`--pretty=oneline --abbrev-commit`"

`--graph`

Affiche le log sous forme de graphe vertical

`--all`

Affiche les logs de toutes les branches

`<since>..<until>`

Affiche les commits entre since et until (voir références)

`<path>...`

Affiche les commits ayant modifié les différents path



GIT LOG

- Par défaut `git log` peut être un peu verbeux

```
$ git log  
commit 7cc6ab3e0d5c85af382fb9b82e1f6f78daf5381c  
Author: Zenika <training@zenika.com>  
Date:   Wed Jul 13 15:23:58 2011 +0100
```

A relevant message

```
commit a5805c0b83691d5dd0937840094522bd9b795bc8  
Author: Zenika <training@zenika.com>  
Date:   Wed Jun 22 17:20:32 2011 +0100
```

Second commit

- Les options de formatage permettent d'avoir des informations plus précises, tout en éludant les données inutiles

```
$ git log --oneline  
7cc6ab3 A relevant message  
5805c0 Second commit  
64bf0dd First commit
```



VOIR L'HISTORIQUE D'UN FICHIER

- La commande `git blame` permet de voir le dernier commit ayant modifié chaque ligne d'un fichier

```
git blame [<rev>] <file>
```

<rev>

Annote le fichier au commit rev

<file>

Annote le fichier file

```
$ git blame README
```

```
df185d44 (Zenika 2011-07-21 22:06:38 +0200 1) Première ligne modifiée
0f56dae4 (Alex 2011-07-19 18:45:27 +0200 2) Deuxième ligne
0f56dae4 (Alex 2011-07-19 18:45:27 +0200 3) Et troisième ligne
6dc66b44 (Colin 2011-07-22 10:17:53 +0200 4) Dernière ligne
```



RESTAURER UNE SAUVEGARDE

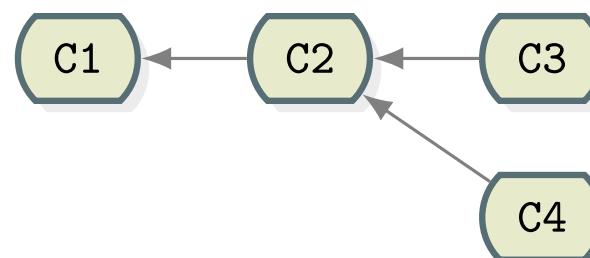
- La commande `git checkout` permet de revenir à l'état de n'importe quel commit effectué

```
git checkout <commit>
```

<commit>

Identifiant ou référence vers le commit à restaurer

- Il est possible après avoir restauré un commit **C2** d'en créer un nouveau appelé **C4** se basant sur **C2**



GIT CHECKOUT

- Pour voir les effets d'un `git checkout` il suffit de regarder le contenu avant

```
$ ls  
FirstFile.txt newFile.txt  
$ git log --oneline  
7cc6ab3 A relevant message  
a5805c0 Second commit  
64bf0dd First commit
```

- Et après

```
$ git checkout a5805c0  
Note: checking out 'a5805c0'.  
You are in 'detached HEAD' state. You can look around, make experimental  
changes and commit them, and you can discard any commits you make in this state  
without impacting any branches by performing another checkout.  
If you want to create a new branch to retain commits you create, you may do so  
(now or later) by using -c with the switch command. Example:
```

```
  git switch -c new_branch_name  
HEAD is now at a5805c0... Second Commit  
$ ls  
FirstFile.txt secondFile.txt
```



GIT RESTORE

- La commande `git restore` permet de faire revenir n'importe quel fichier à l'état de n'importe quel commit effectué

```
git restore [--source commit] <chemin>
```

<commit>

Identifiant ou référence vers le commit à restaurer (optionnel)
index si non précisé

<chemin>

Chemin vers le fichier à restorer

- Cette nouvelle fonctionnalité apporte plus de clarté à cette fonctionnalité
- Encore à l'état d'étude, il se peut que son fonctionnement évolue



QUESTIONS





Lab 2



LES RÉFÉRENCES



PLAN

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- *Les références*
- Utiliser Git en distant
- Configuration et outils externes
- Mettre en place des organisations de travail
- Conclusion
- Projets OpenSource & Pull Requests
- Allons plus loin



LES RÉFÉRENCES

- Le système vu avec la commande `git checkout` a le désavantage de nécessiter de connaître les SHA-1 de chaque commit
- Les références permettent de mettre des noms sur les commits
- Il existe plusieurs type de références
 - **HEAD**, qui représente le commit sur lequel le projet est basé
 - Les **branches**, ce sont des références qui se déplacent lorsqu'on fait un nouveau commit
 - Les **tags**, réfèrentent un commit spécifique et ne bougent pas
 - Le **stash**, qui référence des commits utilisés en interne
 - Les **branches externes**, qui sont les références publiées sur un dépôt externe



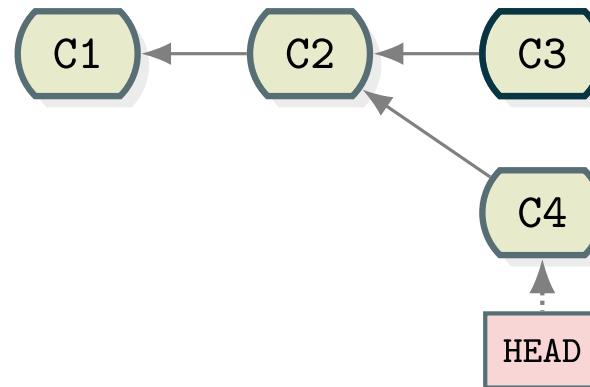
LA RÉFÉRENCE MAGIQUE HEAD

- **HEAD** référence toujours le commit sur lequel les modifications sont basées
 - **HEAD** peut référencer un commit directement (detached head) ou une autre référence
 - En cas de `git checkout`, **HEAD** référence l'élément spécifié



CAS PARTICULIER : DANGLING & DETACHED HEAD

- Un commit doit être référencé ou être le parent d'un autre commit
 - Sinon il est considéré comme **dangling** et est éligible pour une suppression (complète) dans le futur, ici **C3**



LES BRANCHES

- Un concept important, parfois évité par les utilisateurs d'anciens VCS, est remis au goût du jour : les **branches**
- Certains VCS se basent sur le principe qu'une branche n'est qu'un dossier dans lequel on recopie tout l'ancien projet
 - L'historique n'est pas/plus accessible
 - Le changement de branche signifie qu'il faut tout re-télécharger
 - Il est quasi impossible de retracer l'avancement d'une branche sur une période
 - Beaucoup de développeurs préfèrent se risquer à commiter du code "sale" au lieu de passer par l'enfer de la gestion des branches



POURQUOI DES BRANCHES

L'objectif est de permettre de travailler parallèlement sur le même ensemble de fichiers.



GIT SWITCH

- La commande `git switch` permet de se déplacer sur n'importe quel branch ou d'en créer

```
git switch [-c] <branch>
```

-c

Crée une nouvelle branche

<branch>

Nom de la branche cible

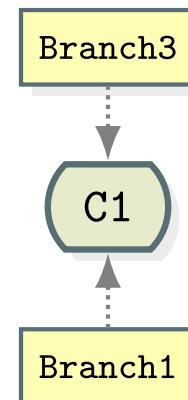
- Cette méthode doit remplacer, à terme, l'utilisation de `git checkout` pour la navigation entre branches
- Encore à l'état d'étude, il se peut que son fonctionnement évolue



LES BRANCHES 1/4

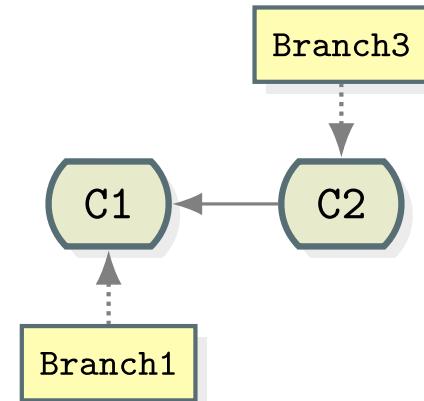
- Pour une visualisation agréable de ces concepts allons sur [Learn Git Branching](#)
- Les branches sont des références **évoluant** lors du commit

Commençons par créer 2 branches, **Branch1** et **Branch3**



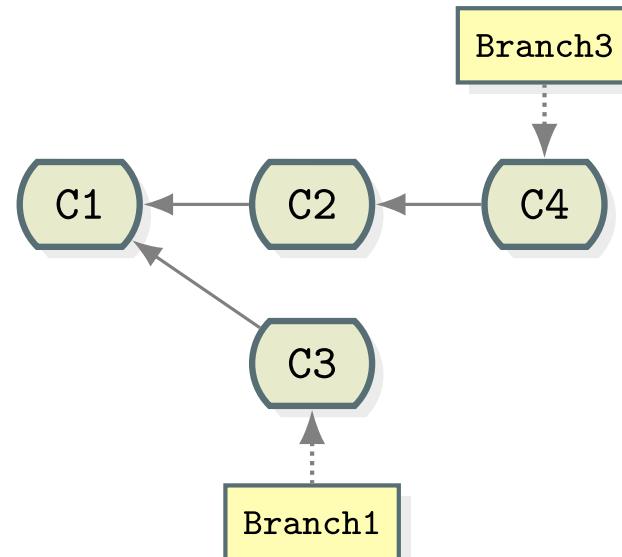
LES BRANCHES 2/4

- Maintenant faisons un commit dans **Branch3**.



LES BRANCHES 3/4

- Maintenant faisons un commit dans **Branch1** et **Branch3**.



LES BRANCHES 4/4

- Si nous créons une branche **Branch2** à partir de **Branch1**, et que nous faisons le commit **C5**, **B2** référencera **C5**.
- Si **HEAD** référence **Branch2** (qui référence **C3**), après le commit de **C5**, **Branch2** référencera **C5**. **HEAD** continuera de pointer vers **Branch2**
- Si **HEAD** référence une branche, cette branche est appelée la **branche courante**



GIT BRANCH

- `git branch` liste les branches locales ainsi que la position de **HEAD**

```
$ git branch
* (detached from a5805c0)
  master
```

- La création d'une branche peut se faire de deux façons

```
$ git branch firstBranch
$ git checkout 64bf0dd -b secondBranch
Switched to a new branch 'secondBranch'
$ git branch
  firstBranch
  master
* secondBranch
```

- L'option **-d** permet de supprimer une branche

```
$ git branch -d firstBranch
Deleted branch firstBranch (was 64bf0dd).
```



RÉSULTAT DE L'OPÉRATION

- Le moyen le plus simple pour voir l'état des branches/commits référencés (non dangling) est de passer par la commande `git log` en spécifiant quelques options
 - `--graph` pour une meilleure visibilité graphique
 - `--all` pour travailler avec toutes les branches/références
 - `--oneline` pour garder le log concis
 - `--decorate` pour afficher les références existantes

```
$ git log --graph --all --oneline --decorate
* b3d9655 (HEAD, master) Merge branch 'secondBranch'
|\ \
| * dc1ce86 (secondBranch) Foo bar modification
* | 7cc6ab3 A relevant message
* | a5805c0 Second commit
| /
* 64bf0dd First commit
```

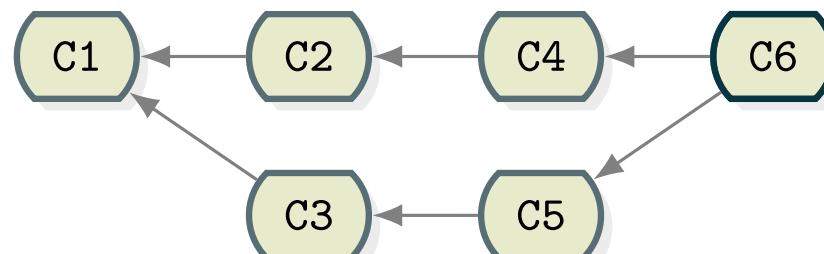




TP3.1 ⇒ TP3.3

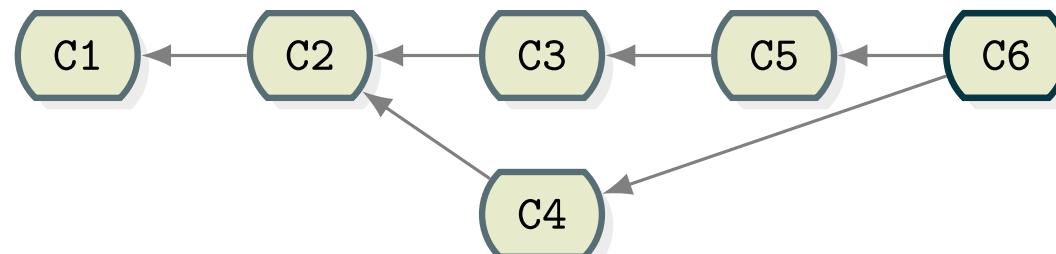
ON FUSIONNE DEUX DÉVELOPPEMENTS

- Le système de commits de Git permet de fusionner deux branches (dans le sens "suite de commits") différentes
 - `git merge` crée un nouveau commit ayant pour parents les deux branches fusionnées



LES MERGES

- Les branches ne sont que des commits distincts (non successifs) sur un graphe
- Merger signifie simplement créer un commit regroupant les deux (ou plus) branches

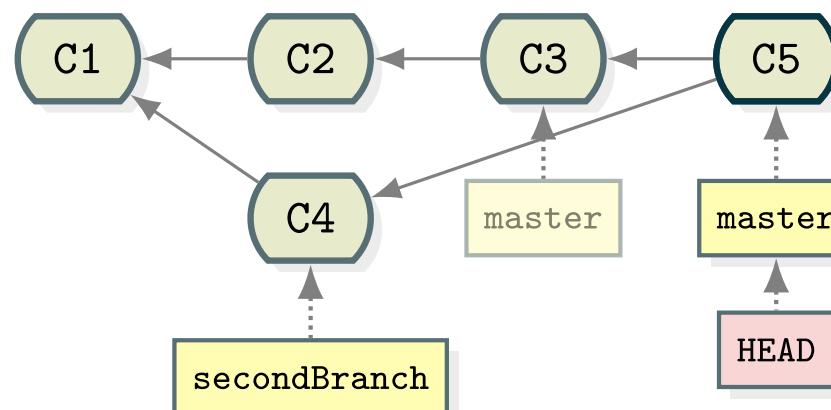


- Une réponse simple à un concept simple
- Des outils vont automatiser le merge et résoudre la plupart des conflits
- Attention cependant, il faudra en résoudre certains soi-même, manuellement



GIT MERGE

- L'opération `git merge` prend en paramètre un (ou plusieurs) commit à fusionner avec le **HEAD** actuel
- Dans notre cas, nous sommes sur **master** et nous voulons merger **secondBranch**. Le résultat du merge est un nouveau commit (**C5**) avec les deux modifications.



GIT MERGE

```
$ git merge secondBranch
Auto-merging firstFile.txt
CONFLICT (content): Merge conflict in firstFile.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
Unmerged paths:
  (use "git add <file>..." to mark resolution)
```

both modified: firstFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

```
$ cat firstFile.txt
<<<<<< HEAD
Hello world!
Test
=====
Foo bar
>>>>> secondBranch
```



RÉSOLUTION DE CONFLITS

Bien que le système de merge de Git soit puissant, il n'est pas impossible d'avoir des conflits.

Le processus à suivre en cas de conflit :

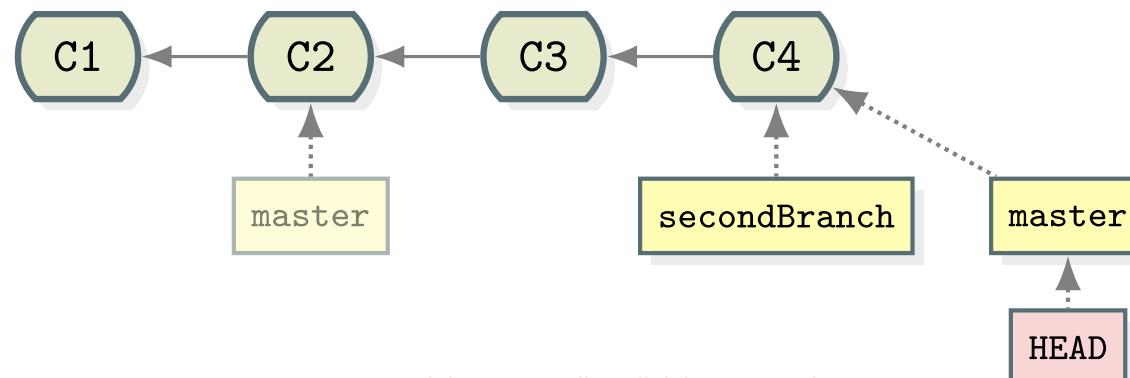
1. Appel de `git merge`
2. En cas de conflit trouver le fichier problématique (`git status`)
3. Résoudre les conflits du fichier (manuellement ou avec `git mergetool`)
4. Ajouter le fichier résolu à l'index (`git add`), si d'autres fichiers sont en conflit retourner à l'étape 2
5. Commiter (`git commit`), un message de commit sera proposé par défaut.

Il est recommandé de documenter la résolution des conflits



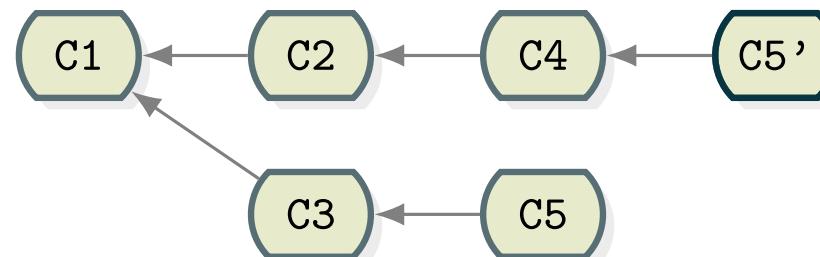
FAST FORWARD

- Dans le cas où **master** n'a pas divergé depuis la création de **secondBranch**
 - Le merge de **C3** et **C4** dans **C2** aurait pointé vers le même tree que **C4**
 - Pour éviter des opérations futiles, Git détecte ce cas et le gère en **fast-forward**
- Le **fast-forward** consiste lors d'un merge, si le **HEAD** est un ancêtre direct de l'autre branche, à simplement déplacer celui-ci sur la branche



UNE AUTRE MANIÈRE DE FUSIONNER

- Il est possible de rejouer des commits à partir d'un autre commit
 - `git rebase` "déplace" et modifie une suite de commits



REBASE DE COMMITS

- Le but de la commande `git rebase` est de pouvoir facilement déplacer/manipuler un ensemble de commits et la référence associée

```
git rebase [-i] [--onto <destination>] <from> [<to>]
```

`-i` Mode interactif

`--onto <destination>`

Commit sur lequel les commits vont être déplacés

Si non spécifié la valeur de `<from>` est utilisé

`<from>`

Commit à partir du quel les commits à déplacer sont listés

Tous les commits présents dans la branche `<to>` et

non présents dans la branche de `<from>` sont pris en compte

`<to>`

Commit/référence déplacée

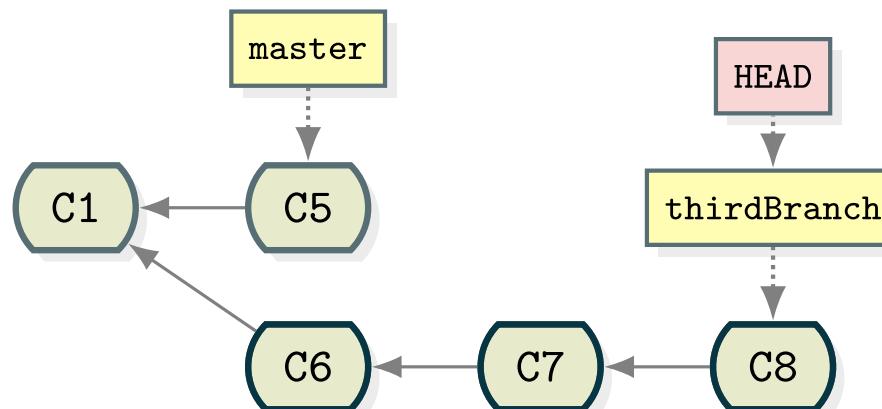
Si non spécifié, HEAD est utilisé

- Le mode interactif permet une manipulation plus précise des données, alors que le mode classique se contente de "copier" les commits



AVANT LE REBASE

- Pour effectuer un rebase, une branche avec quelques commits est créée



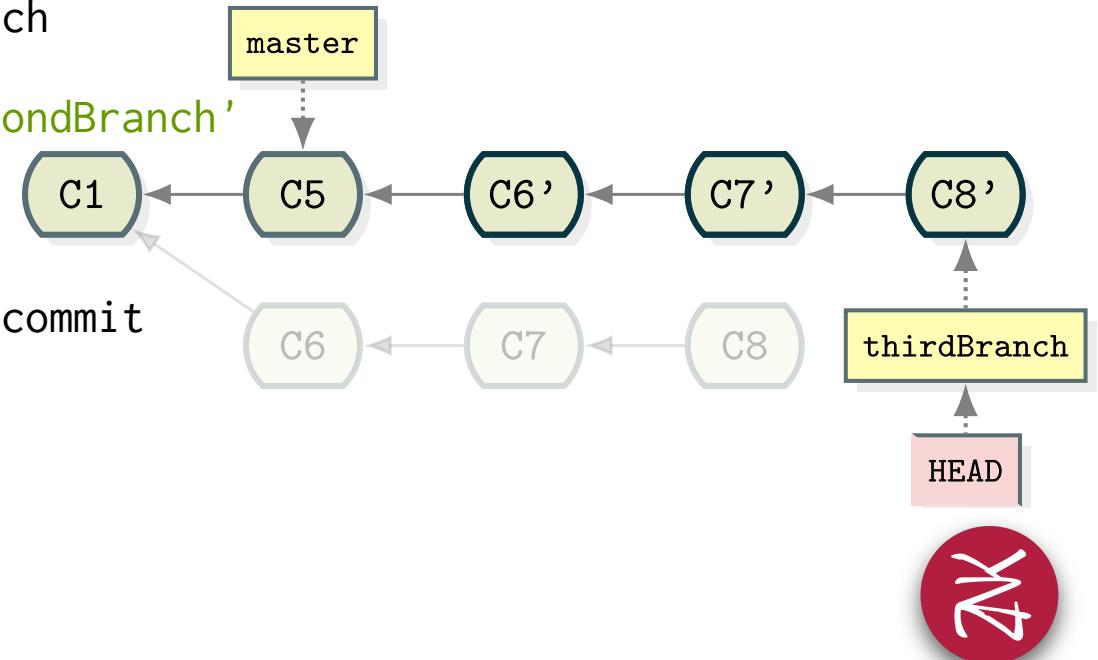
```
$ git log --graph --all --oneline --decorate
* 1ce14e2 (HEAD, thirdBranch) Third commit on thirdBranch (enough !)
* 041fa9c Second commit on thirdBranch
* fb0aed3 ThirdBranch creation
| * 041fa9c (secondBranch) Second commit
| | * b3d9655 (master) Merge branch 'secondBranch'
| |
| / \
| |
| |
| * | dc1ce86 Foo bar modification
| / /
| * 7cc6ab3 A relevant message
| * a5805c0 Second commit
|
* 64bf0dd First commit
```



GIT REBASE

- L'opération de rebase se fait en une ligne

```
$ git rebase --onto master master thirdBranch
First, rewinding head to replay your work on top of it...
Applying: ThirdBranch creation
Applying: Second commit on thirdBranch
Applying: Third commit on thirdBranch (enough !)
$ git log --graph --all --oneline --decorate
* d29a5fc (HEAD, thirdBranch) Third commit on thirdBranch (enough !)
* 39aeca1 Second commit on thirdBranch
* fc1f274 ThirdBranch creation
* b3d9655 (master) Merge branch 'secondBranch'
  \
  * 7cc6ab3 A relevant message
  * a5805c0 Second commit
  * 041fa9c (secondBranch) Second commit
  /
  * dc1ce86 Foo bar modification
  /
* 64bf0dd First commit
```



EN CAS DE CONFLITS

- Le fait de ré-appliquer plusieurs commits mènera sûrement à des conflits. `git rebase` les gère d'une façon particulière
 - Un commit est appliqué, en cas de conflit, le rebase s'arrête
 1. L'utilisateur fixe le conflit et exécute
`git rebase --continue`
 2. L'utilisateur n'applique pas ce commit et exécute
`git rebase --skip`
 3. L'utilisateur arrête le rebase et revient à l'état initial grâce à
`git rebase --abort`
- De cette manière, tous les conflits sont corrigés dans le commit les générant



COPIE DE COMMIT

- Il est possible de copier un commit vers un nouvel emplacement
 - L'opération consiste à récupérer les modifications apportées par un ou plusieurs commits et les rejouer sur le nouvel emplacement, avec les mêmes messages
 - Les anciens commits sont toujours présents
 - Les nouveaux commits (les copies) ont leur propres hash
- L'opération **git cherry-pick** copie un commit vers le nouvel emplacement, l'ancien commit n'est pas affecté
- **git rebase** copie une suite de commits vers le nouvel emplacement, le résultat est que la référence vers les anciens commits est déplacée
 - Les anciens commits sont donc potentiellement dangling

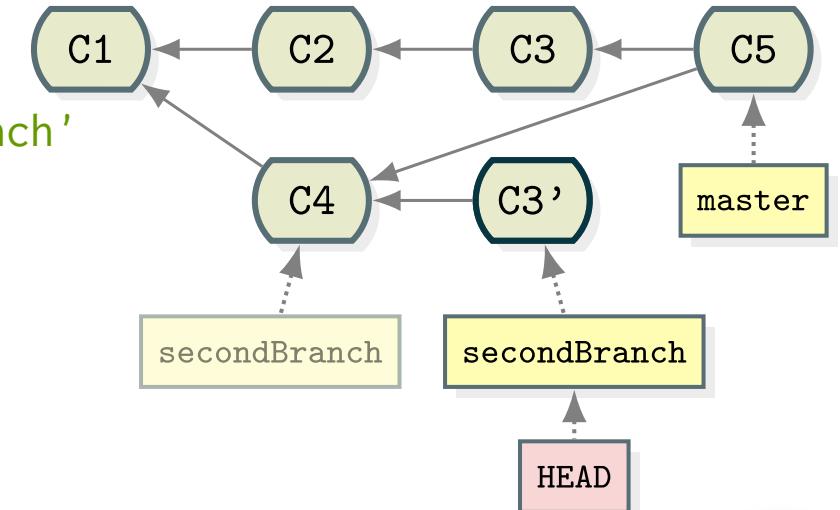


GIT CHERRY-PICK

- `git cherry-pick` permet de récupérer les changements apportés par un commit et les reproduire (ici duplication de **C3**)

```
$ git cherry-pick a5805c0
[secondBranch 041fa9c] Second commit
 2 files changed, 0 insertions(+), 1 deletions(-)
 delete mode 100644 firstDir/firstFileCopy.txt
 Rename firstDir/secondFile.txt => secondFile.txt (100%)

$ git log --graph --all --oneline --decorate
* 041fa9c (HEAD, secondBranch) Second commit
| * b3d9655 (master) Merge branch 'secondBranch'
| |\ \
| | /
| |
* | dc1ce86 Foo bar modification
| * 7cc6ab3 A relevant message
| * a5805c0 Second commit
|
* 64bf0dd First commit
```





TP3.4 ⇒ TP3.6

MODE INTERACTIF

- Le mode interactif ouvre un éditeur dans lequel il est possible de choisir pour chaque commit la façon dont il sera géré

```
pick 48c7571 FourthBranch creation
pick 98f5833 Second commit in fourthBranch

# Rebase b3d9655..98f5833 onto b3d9655
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```



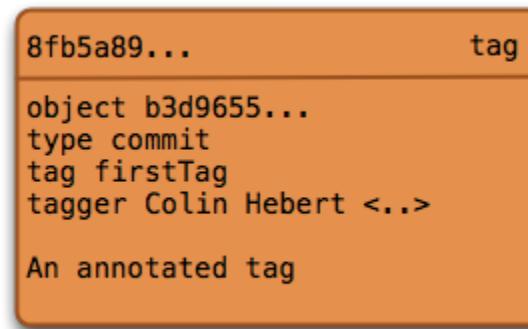
D'AUTRES RÉFÉRENCES/OBJETS

- En plus des références **HEAD** et des branches, il est possible de créer des références "fixes" vers certains commits, des **tags**
- Le but est de pouvoir identifier facilement un commit particulier, par exemple une release d'un produit
- Il existe deux types de tags
 - Les **tags simples**, ce sont des références basiques vers des commits
 - Les **tags annotés/signés**, ces tags contiennent des informations supplémentaires
 - Créeur du tag (nom et email)
 - Date de création du tag
 - Message de description
 - Signature/Checksum



TAGS ANNOTÉS

- Le système de tag annoté requiert d'enregistrer un nouvel objet dans la base interne de Git



```
$ git cat-file -p 8fb5a89
Object b3d9655...
type commit
tag firstTag
tagger Zenika <training@zenika.com> Mon Jul 25 16:23:22 2011 +0100
An annotated tag
```

- Seul le tag annoté est enregistré ainsi. Un tag standard est juste compté comme une référence
- Un tag (de n'importe quel type) pointe vers un **object** et non spécifiquement vers un commit. Il peut donc pointer vers un **tree** ou même un **blob**



CRÉATION D'UN TAG

- La commande pour créer et manipuler les tags est `git tag`

```
git tag [-a|-s|-d|-v|-l [-n]] [-m <msg>] <tagname> [<object>]
```

- a Crée un tag annoté non signé
- s Crée un tag annoté signé grâce à une clef GPG
- d Supprime un tag donné
- v Vérifie la signature d'un tag annoté signé
- l Liste les tags ayant un nom contenant <tagname>
- n Si -l est activé, affiche la première ligne du message du tag

Si aucune de ces options n'est utilisée un tag "simple" sera créé

`-m <msg>`

Spécifie le message lors de la création d'un tag annoté

`<tagname>`

Nom du tag à créer/supprimer/vérifier

`<object>`

Élément à tagger. Si non spécifié le commit de HEAD est utilisé

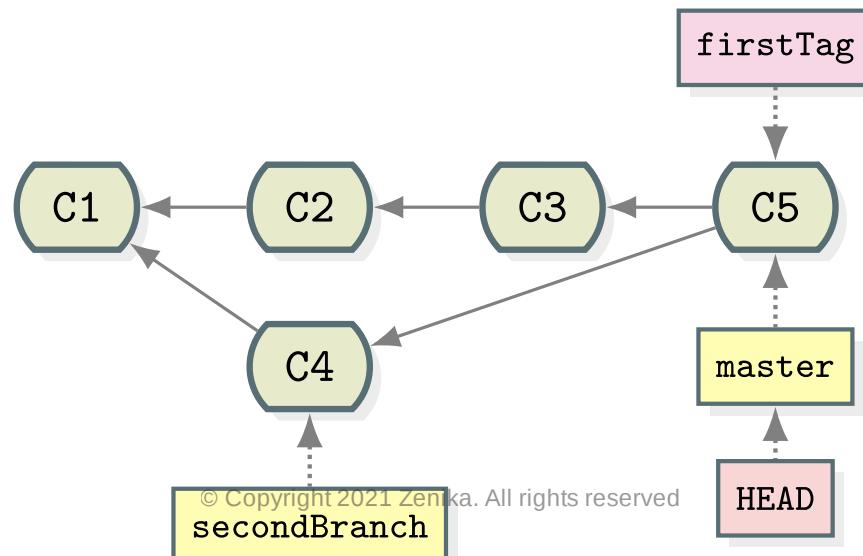
- Le tag, une fois créé, ne pourra pas être modifié et pointera définitivement vers le même objet



GIT TAG

- Voici le résultat d'une création de tag

```
$ git tag -a -m "An annotated tag" firstTag
$ git log --graph --oneline --decorate
* b3d9655 (HEAD, tag: firstTag, master) Merge branch 'secondBranch'
|\ \
| * dc1ce86 Foo bar modification
* | 7cc6ab3 A relevant message
* | a5805c0 Second commit
|/
* 64bf0dd First commit
$ git tag -n -l firstTag
FirstTag      An annotated tag
```





TP3.7

LE STASH

- Comment changer de branche sans perdre son travail en cours ?
- Une solution de stockage temporaire existe, le ***stash***



MANIPULER LE STASH

- La commande pour créer et manipuler le stash est `git stash`

```
git stash [save | list | (apply | drop | pop | show) <stash>]
```

save

Crée un stash à partir de l'état actuel du dépôt

list

Liste tous les stash existants

apply

Applique le stash <stash>

drop

Supprime le stash <stash>

pop

Applique et supprime le stash <stash> (apply suivit de drop)

show

Affiche les changements apportés par le stash <stash>

Si aucune de ces commande n'est utilisée, "save" sera employé

<stash>

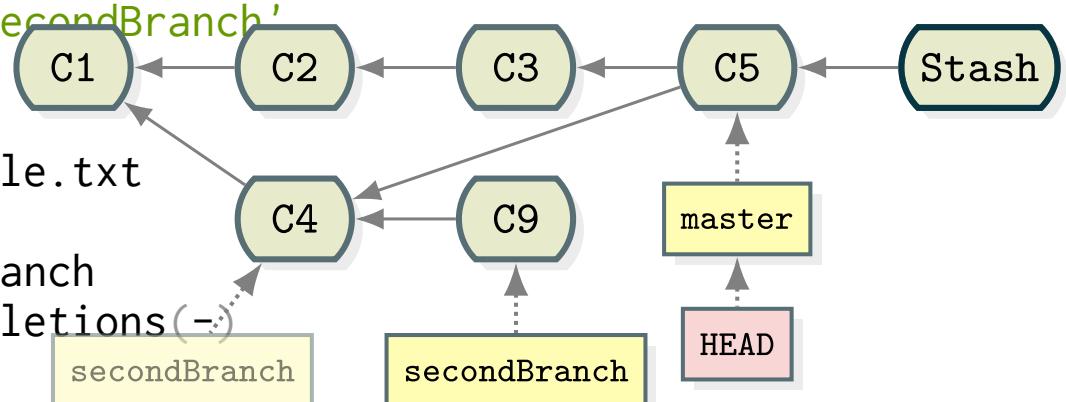
Stash sur le quel l'opération doit-être effectuée



GIT STASH

- La commande `git stash` en action

```
$ git status -s
$ echo "New line" >> firstFile
$ git status -s
M firstFile.txt
$ git stash
Saved working directory and index state WIP on master: b3d9655 Merge branch
'secondBranch'
HEAD is now at b3d9655 Merge branch 'secondBranch'
$ git checkout secondBranch
Switched to branch 'secondBranch'
$ echo "Fix code" >> firstDir/secondFile.txt
$ git ci -a -m "Fixing secondBranch"
[secondBranch 848277f] Fixing secondBranch
 1 file changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch 'master'
$ git stash pop -q
$ git status -s
M firstFile.txt
```



INFORMATIONS SUR GIT STASH

- Dans le cadre d'un travail en équipe, les stashes sont strictement personnels, ils ne sont pas partagés
- Il est possible de faire appel aux stashes précédents grâce à **stash@{N}** avec **N** le numéro du stash
 - Le dernier stash créé aura toujours le numéro **0**
- Les fichiers non versionnés ne sont pas ajoutés au stash
- Un stash **pop** ou **drop** supprime le stash, en cas de problème les données ne sont pas perdues, certaines techniques permettent de retrouver les données
- Il est possible d'appliquer le contenu d'un stash n'importe où
- Lorsqu'un stash est appliqué il peut y avoir des conflits auquel cas il faudra merger le résultat





Lab 4

RÉFÉRENCES AVANCÉES

- Pour augmenter la facilité de navigation dans Git, il est possible de manipuler les références avec des informations supplémentaires
 - `<ref>~<n>` représente le Nième ancêtre de `ref`
 - `<ref>^<n>` représente le Nième parent de `ref` (dans un merge)
 - Si `n` n'est pas spécifié sa valeur est `1`
- `HEAD^ == HEAD~`
- `HEAD^^ == HEAD~~ == HEAD~^ == HEAD^~ == HEAD~2`
- `HEAD^0 == HEAD~0 == HEAD`
- `HEAD^1^1^2 == HEAD~2^2`
- Ces références sont pratiques pour remonter dans l'historique sans avoir à connaître les hashes



ENSEMBLES DE COMMITS

- Il est possible pour des commandes telles que `git log` de spécifier un ensemble de commits à prendre en compte
 - `<ref>` Tous les ancêtres de `ref` plus `ref`
 - `<ref1> <ref2>` Tous les ancêtres de `ref1` et ceux de `ref2` plus `ref1` et `ref2` (Union \cup)
 - `^<ref1> <ref2>` Tous les ancêtres de `ref2` sauf ceux qui sont aussi parents de `ref1` (Différence \setminus)
 - `<ref1>..<ref2>` Idem
 - `<ref1>...<ref2>` Tous les ancêtres que `ref1` et `ref2` n'ont pas en commun plus `ref1` et `ref2` (Différence symétrique Δ)
 - `<ref>^@` Tous les ancêtres de `ref` (mais pas `ref`)
 - `<ref>^! ref` mais pas ses ancêtres







UTILISER GIT EN DISTANT



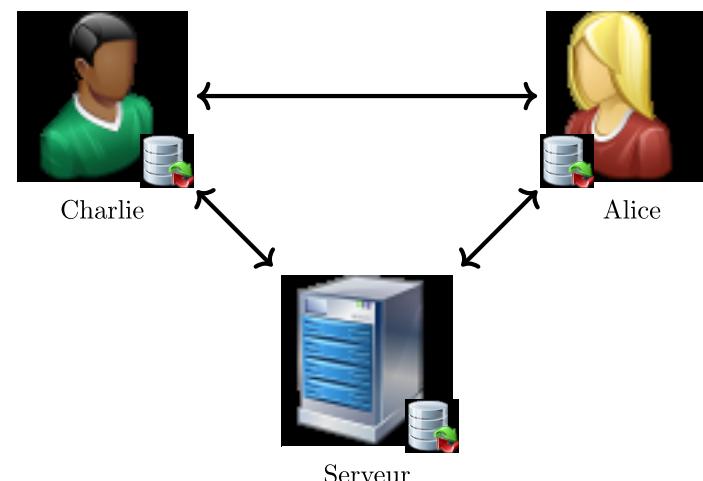
PLAN

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- *Utiliser Git en distant*
- Configuration et outils externes
- Mettre en place des organisations de travail
- Conclusion
- Projets OpenSource & Pull Requests
- Allons plus loin



ACCÈS DISTANT

- Les systèmes de VCS sont souvent synonymes d'échanges avec un dépôt distant
- Git possède cette capacité à se connecter vers un ou plusieurs dépôts pour échanger les différentes modifications qui ont eu lieu



TYPES D'ACCÈS

- Les dépôts sont accessibles au travers de différents protocoles
 - File system : Utilisé jusqu'à présent, tout le monde ayant les droits en écriture sur le dossier peut de fait modifier le dépôt
 - SSH : Sécurisé et le plus commun, il permet de faire une gestion fine des droits d'accès en distant
 - git : Embarqué avec Git, similaire au SSH mais sans authentification
 - HTTP(s) : Protocole autorisé par la plupart des pare-feux, il est souvent utilisé en lecture seule pour publier un projet



UTILISATION D'UN DÉPÔT DISTANT

- Le système proposé par Git pour la gestion des dépôts distants fonctionne sur le modèle
 - Déclaration du dépôt (chemin d'accès + nom)
 - Récupération des données du dépôt
 - Branches
 - Tags
 - Objets Git non "dangling"
 - Envoi des nouvelles données, si les droits en écriture sont activés



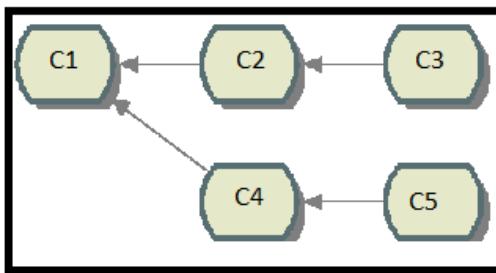
COMMENT COLLABORER AVEC UN DÉPÔT DISTANT

- `git remote`
- `git clone`
- `git pull`
- `git push`



QUELQUES EXPLICATIONS AVANT LA THÉORIE

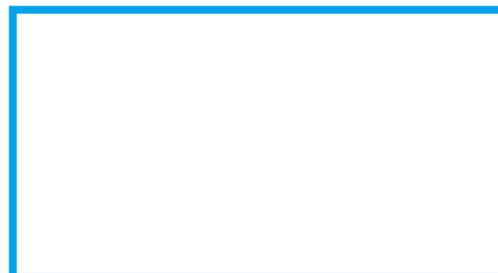
Nous avons un dépôt distant, avec un historique.



Remote: origin



Remote



Historique Local



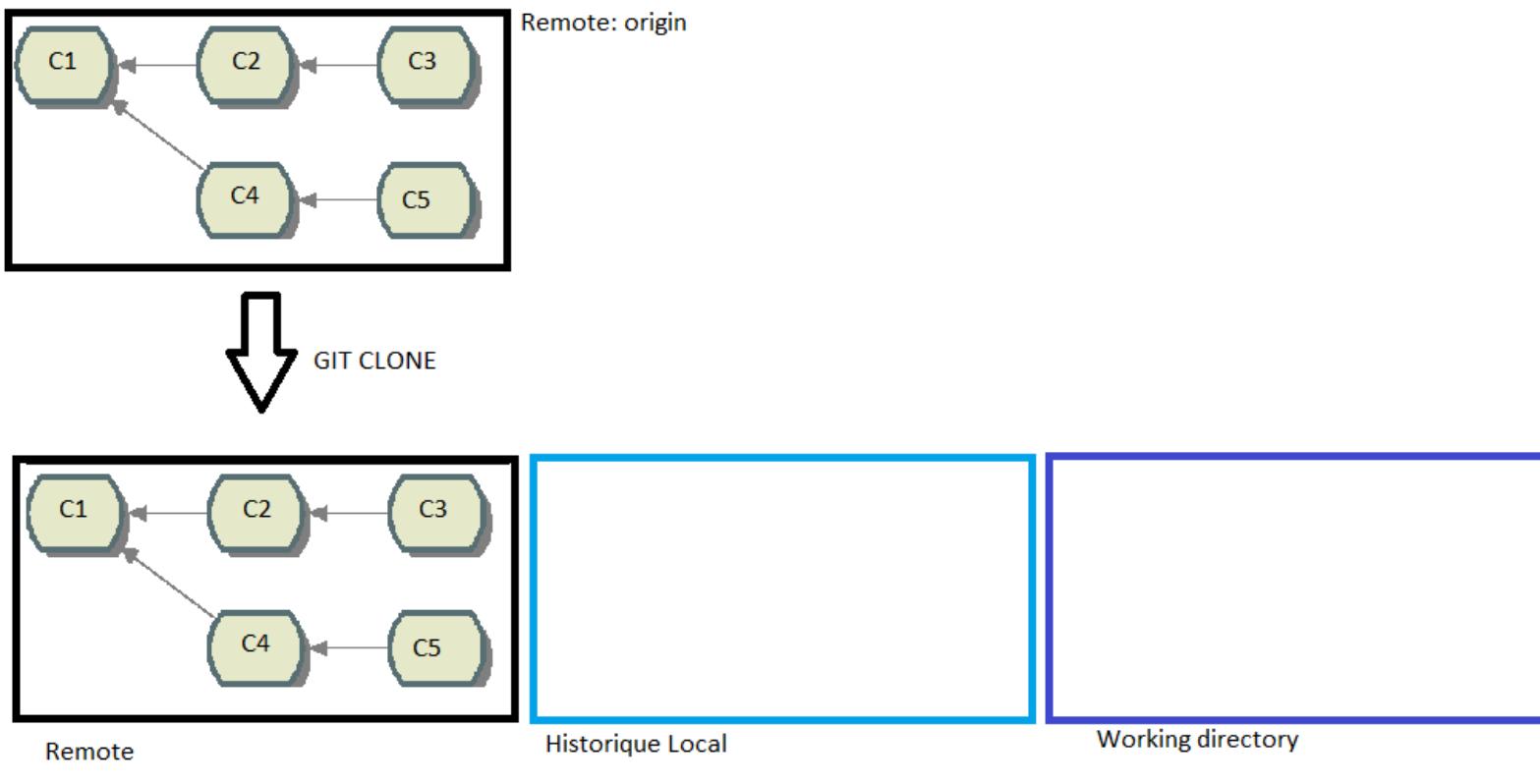
Working directory

Repository GIT Local



QUELQUES EXPLICATIONS AVANT LA THÉORIE

Lors du clone, nous récupérons cet historique localement.



Repository GIT Local

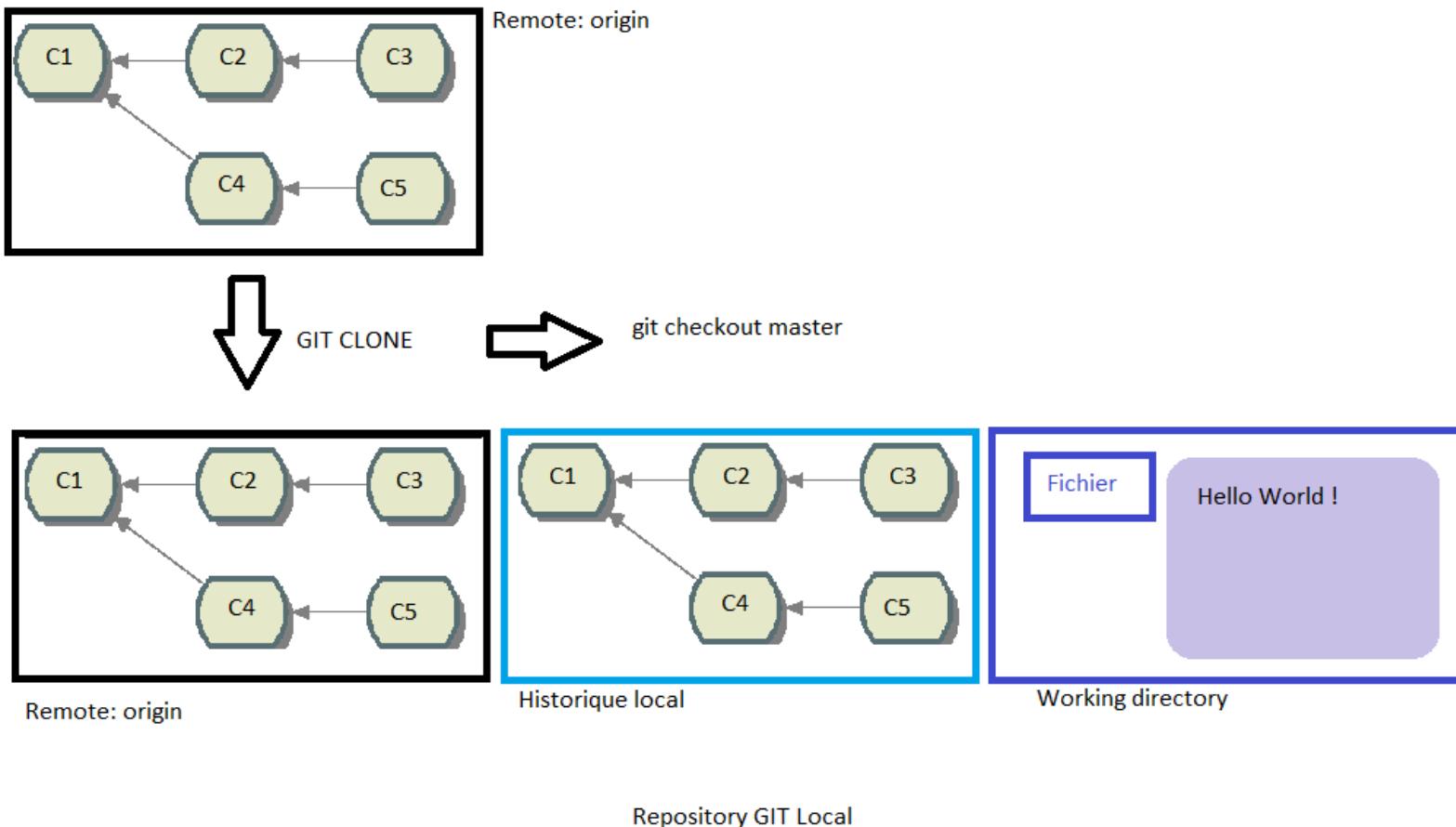


LE CLONE FAIT AUSSI UN CHECKOUT DE LA BRANCH MASTER PAR DÉFAUT

Cela a pour effet de remplir votre répertoire de travail avec les fichiers.

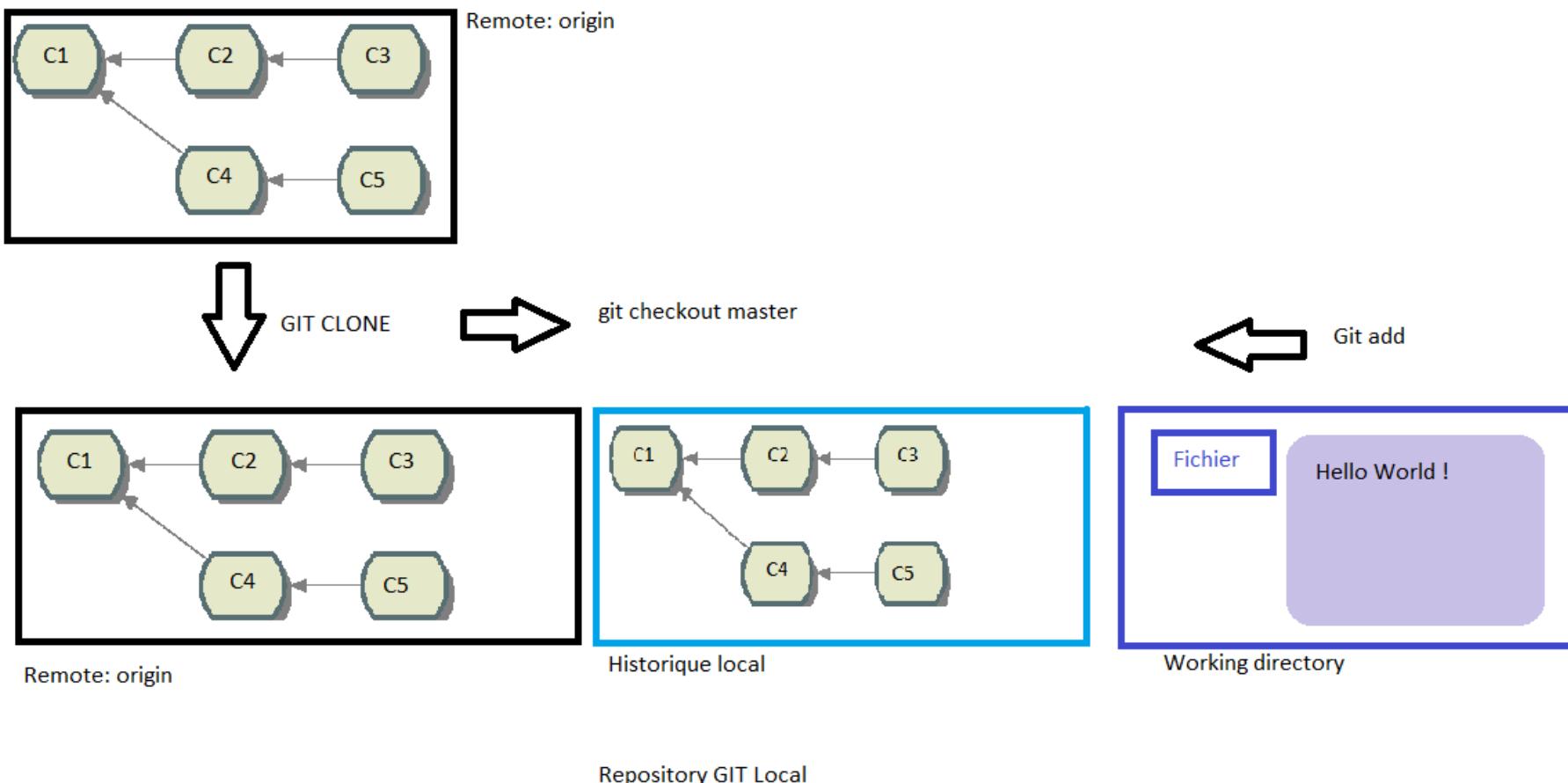


MODIFICATIONS MAINTENANT LE FICHIER



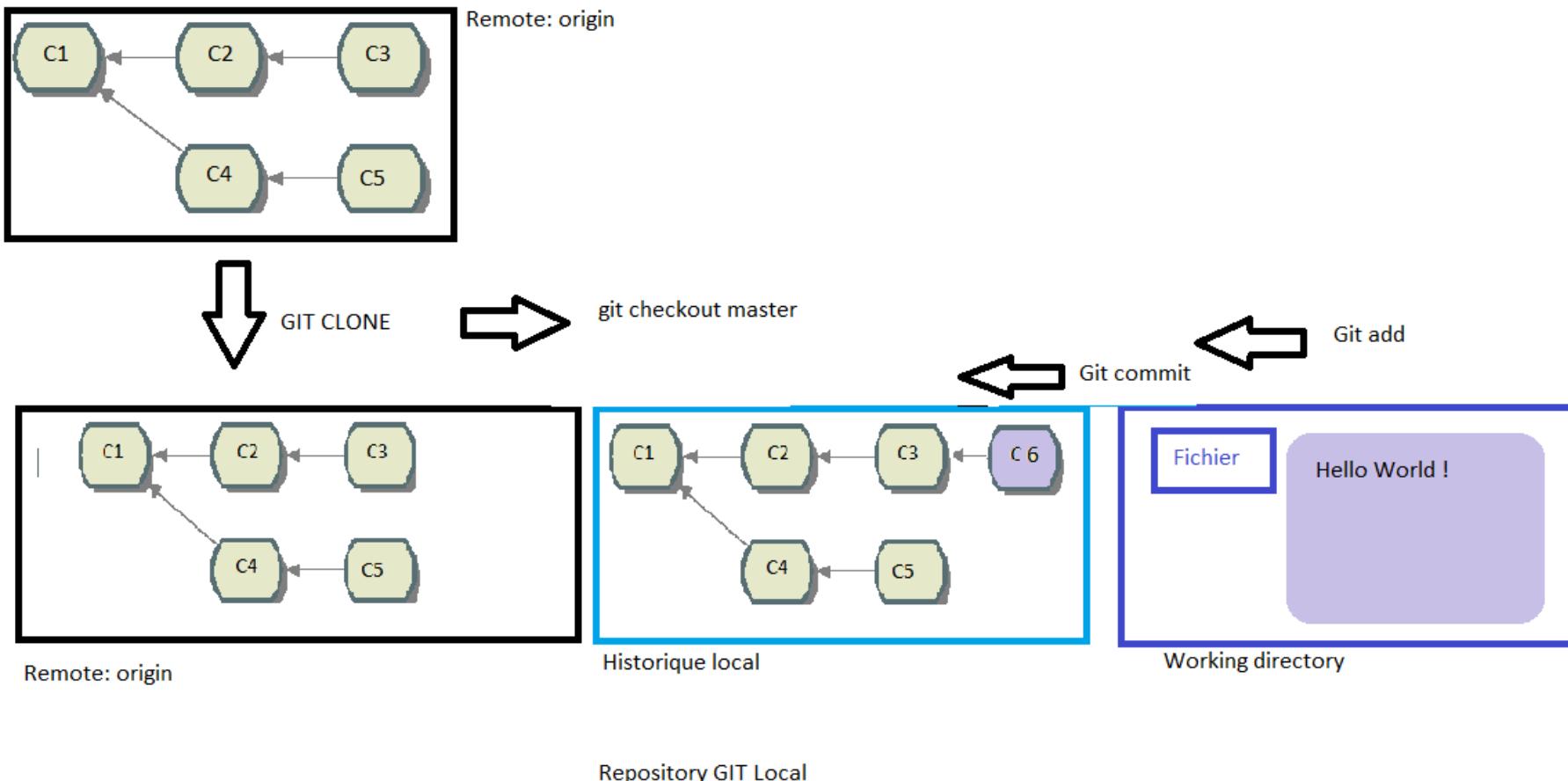
FAISONS UN ADD

Rappel : cela modifie l'index local, mais pas votre vision du remote, ni le remote.

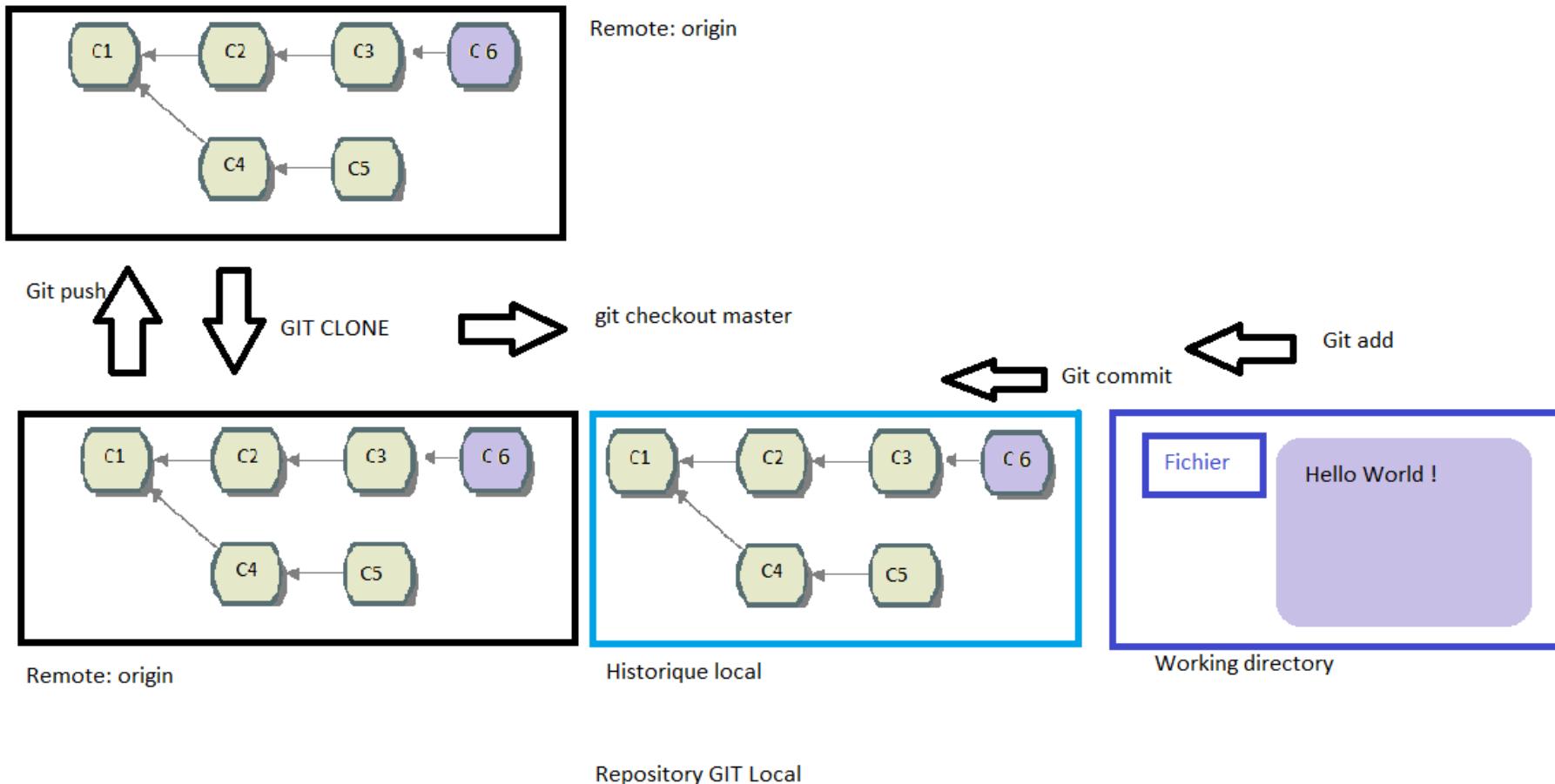


FAISONS UN COMMIT

Le commit n'a qu'une portée locale.



ET MAINTENANT FAISONS UN PUSH



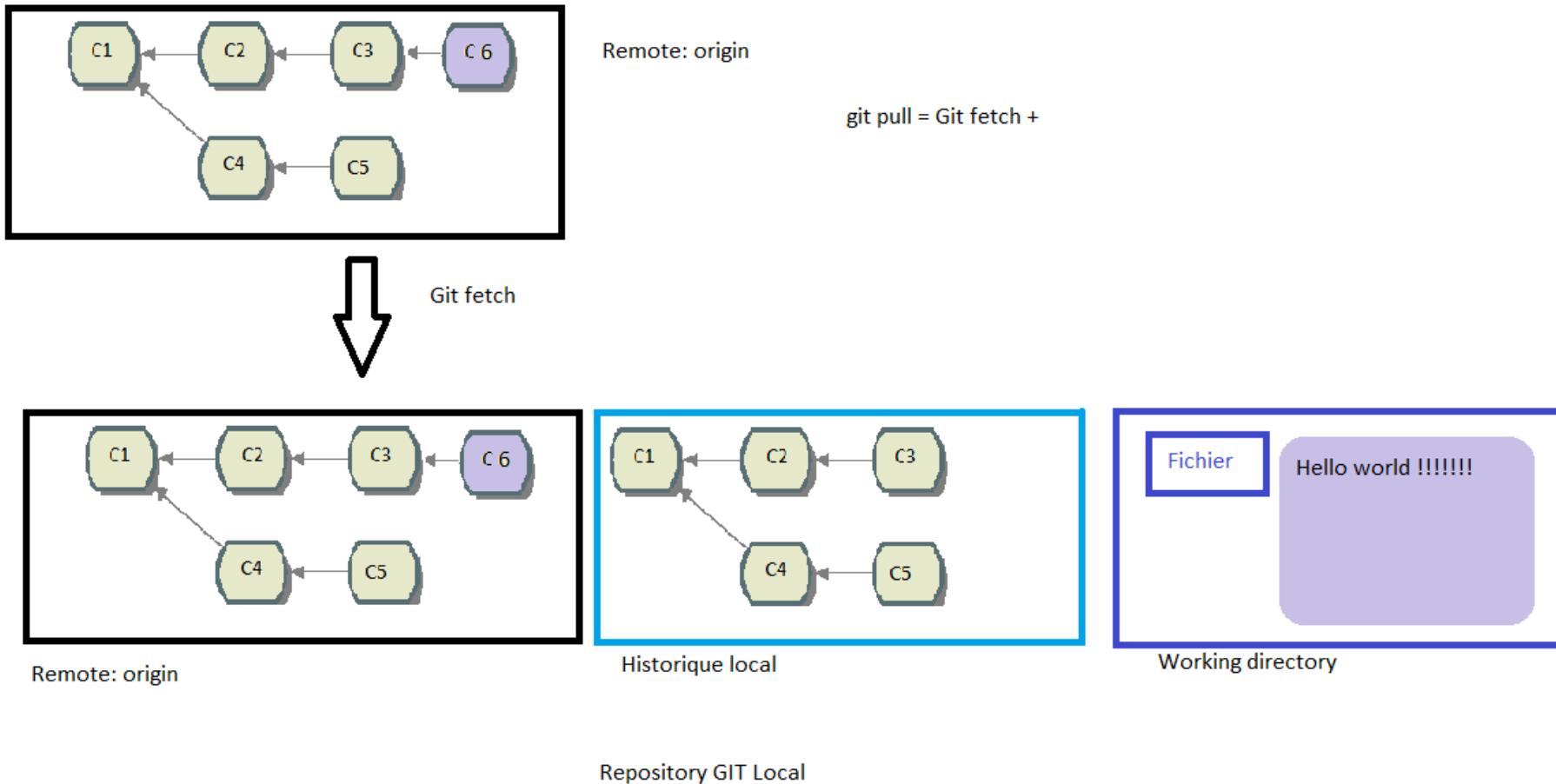
AUTRE CAS, MAINTENANT, JE VEUX AUSSI RÉCUPÉRER LE TRAVAIL

Je ne peux pas faire un `git clone`, je ne veux pas tout récupérer. Je fais un `git pull`.

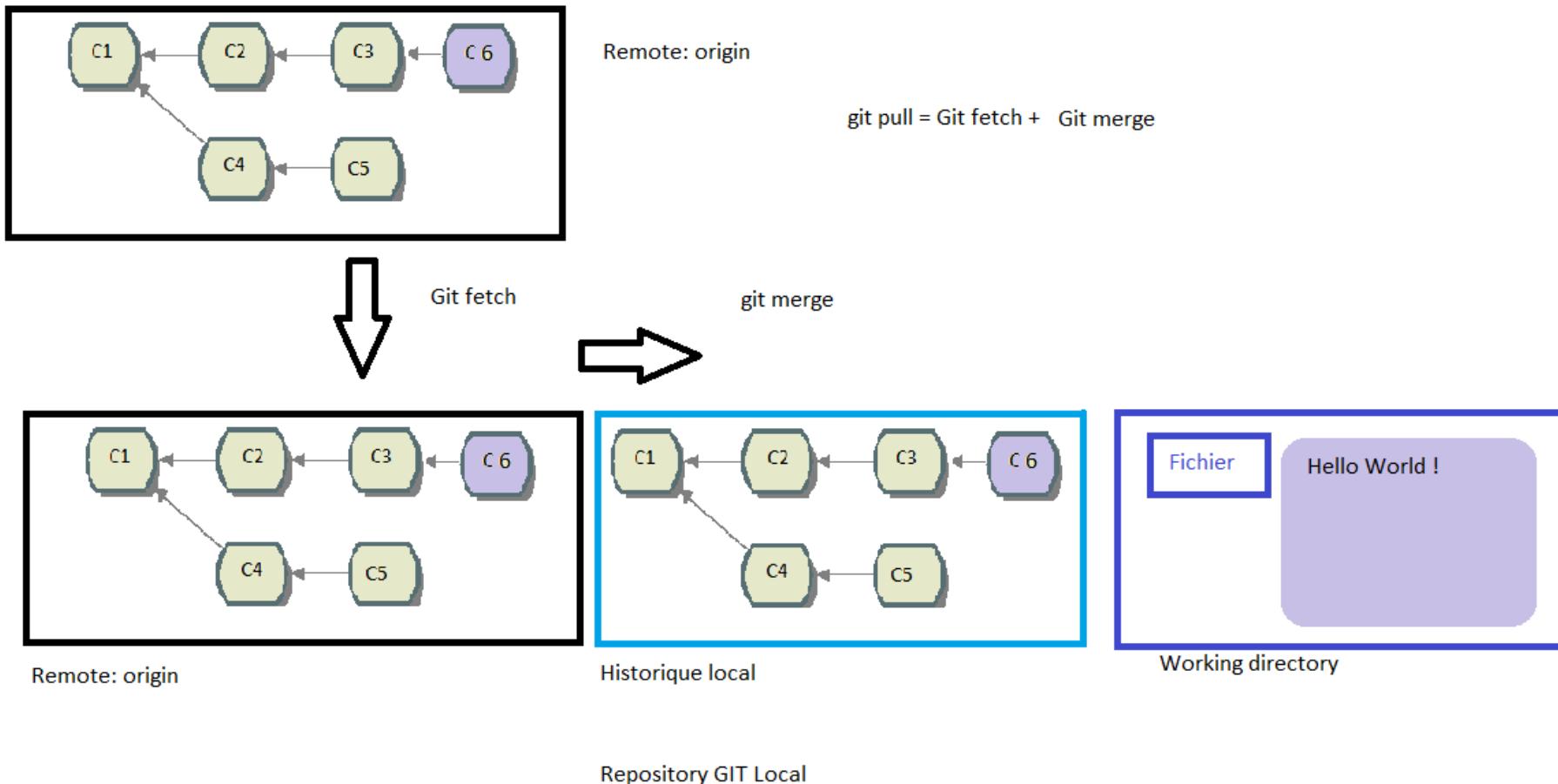


PULL : FETCH + ??

Le **pull** fait d'abord un **fetch** : copie locale du graphe distant.



PULL : FETCH + MERGE

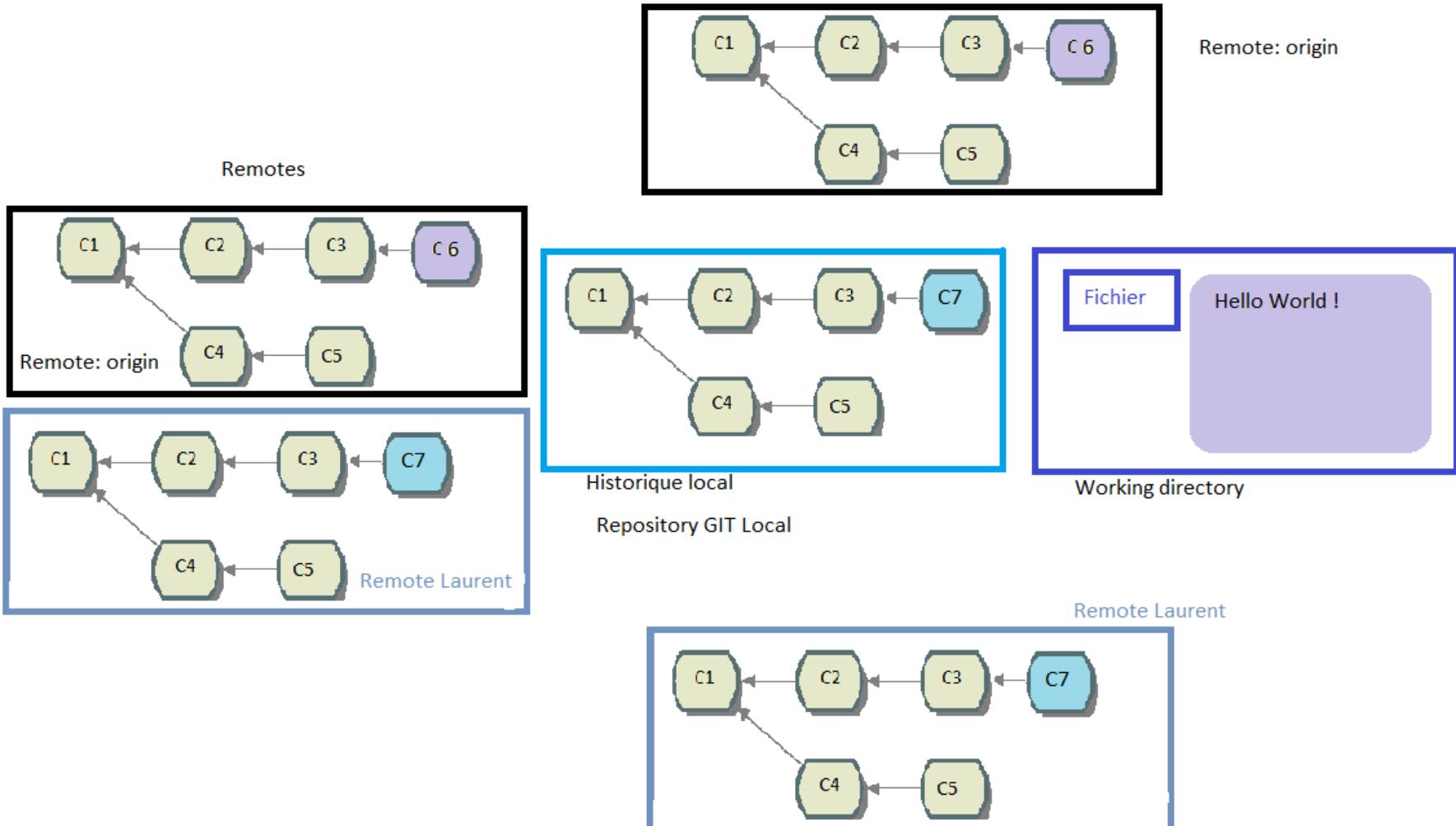


C'est pour cela que nous pouvons avoir des conflits lors d'un **pull**.

© Copyright 2021 Zenika. All rights reserved



ET SI NOUS AVIONS PLUSIEURS RÉPERTOIRES DISTANTS



DÉCLARATION D'UN DÉPÔT EXTERNE

- La déclaration et gestion des dépôts externes se fait grâce à **git remote**

```
git remote [add|rm|show] <nom> [<url>]
```

add

Ajoute le dépôt "nom" pointant vers "url"

rm

Supprime le dépôt externe "nom" de la liste

show

Affiche les informations sur le dépôt "nom"

- Sans aucun argument, **git remote -v** affiche tous les dépôts enregistrés



RÉCUPÉRATION DES DONNÉES

- Pour récupérer les données, `git fetch` est utilisé

```
git fetch [--all|<nom>]
```

--all

Récupère les derniers ajouts sur tout les dépôts

<nom>

Récupère les derniers ajouts sur le dépôt "`nom`"

- Les informations récupérées sont uniquement les références disponibles ainsi que tous les commits associés
- Cette récupération ne nécessite pas de recharger tout le contenu du dépôt externe
- Aucune référence n'est modifiée en local, seuls de nouveaux objets sont ajoutés à la base locale



RÉCUPÉRATION ET INCORPORATION

- Il est aussi possible de charger et incorporer directement les dernières modifications apportées sur une branche
- La commande pour ceci est `git pull`

```
git pull [--rebase] <depot> <branche>
```

`--rebase`

Utilise la technique de rebase au lieu d'un merge

`<depot>`

Dépôt où sont situées les données

`<branche>`

Nom de la branche à récupérer et appliquer sur la branche courante

- La première action faite par `git pull` est la même que `git fetch`, les dernières modifications sont rapatriées
- Après cette récupération Git essaye de fusionner la-dite branche avec la branche actuelle (la copie de travail pointée par `HEAD`)



ENVOI DES NOUVEAUX COMMITS

- Pour partager le travail effectué, il est possible de passer par la commande **git push**

```
git push [--all] <depot> [--delete] [[<brancheL>]:<brancheD>]
```

--all

Crée tous les éléments présents sur le dépôt local, sur le dépôt distant
<depot>

Dépôt externe où envoyer les données

--delete

Supprime la branche distante

<brancheL>

Nom de la branche local à envoyer sur le dépôt externe

<brancheD>

Nom de la branche distante

- Si la branche locale n'est pas spécifiée ou que l'option **--delete** est utilisée, la branche distante est supprimée : **git push --delete branchToDelete** ou **git push :branchToDelete**



RÈGLES DE BIENSÉANCE

- Les possibilités de Git sont quasi-illimitées, seulement il persiste tout de même certaines règles à respecter
- Tout commit fait sur un dépôt public doit (dans la mesure du possible) être final
 - Pas de rebase
 - Pas de suppression de commit
 - Pas de suppression de branche non-fusionnée
 - Pas de modification/suppression de tag
 - Pas d'amendement sur les commits
- De base Git interdira de pusher des commits non **fast forward**
- Dans le cas où ceci serait fait, il faudrait en informer tous les développeurs afin que ceux-ci récupèrent les modifications et rebasent leur travail
- Les dépôts privés cependant ne nécessitent pas cette attention



PARTAGE "MANUEL"

- Il est possible grâce à `git diff` et `git apply` de partager manuellement des patchs
- `git diff` permet de générer des patchs

```
git diff <commit1> [<commit2>]
```

<commit1> <commit2>

Commits à comparer, commit2 est HEAD si non renseigné

- `git apply` applique un patch spécifié à la branche courante

```
git apply <patch> [<patch ...>]
```

<patch>

Fichier contenant le patch à appliquer

- Attention : ce système peut être pratique ponctuellement mais risque de créer des conflits



COPIE PARFAITE

- S'il est possible de créer un dépôt local, se brancher sur un dépôt externe, récupérer son contenu et ses références puis checkout son **HEAD**, Git propose une solution simple, **git clone**

```
git clone <depot> [<dossier>]
```

<depot>

Dépôt externe à cloner

<dossier>

Dossier du projet à créer, par défaut ce sera le nom du dernier élément de l'URL du dépôt

- Git s'occupe donc de tous les appels qui auraient été faits en temps normal
- Le remote par défaut s'appellera **origin**



rendre son dépôt **PUBLIC**

- `git clone --bare`
- Cela permet de construire un dépôt fait pour être cloné. Il ne contient pas de working directory.



QUESTIONS





Lab 6



Lab 7



CONFIGURATION ET OUTILS EXTERNES



PLAN

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- *Configuration et outils externes*
- Mettre en place des organisations de travail
- Conclusion
- Projets OpenSource & Pull Requests
- Allons plus loin



CONFIGURATION GÉNÉRALE

- La configuration se fait avec la commande `git config`

```
git config [--global] [--get] <option>
```

--global

Modifie `~/.gitconfig` au lieu de `.git/config`

--get

Récupère la configuration au lieu de la modifier

<option>

Option à modifier

Généralement sous la forme "groupe.option"

- Deux options ont déjà été créées plus tôt

- `user.name`

- `user.email`

- ***Attention les configurations ne sont pas partagées avec les autres dépôts si l'on omet --global !***



LES ALIASES

- Il est possible de créer ses propres commandes Git
- Ces commandes peuvent-être des raccourcis vers des commandes Git existantes ou bien des scripts entiers
- La création d'un alias **a** consiste en l'ajout d'une configuration **alias.a** dans le fichier de configuration
- Par exemple la commande **git log --graph --oneline --decorate** utilisée pour visualiser les commits peut-être crée sous une forme plus succincte

```
$ git config --global alias.glog 'log --graph --oneline --decorate'  
$ git glog  
* b3d9655 (HEAD, tag: firstTag, remote/master, master) Merge branch  
'secondBranch'  
|\  
| * dc1ce86 Foo bar modification  
* | 7cc6ab3 A relevant message  
* | a5805c0 Second commit  
|/  
* 64bf0dd First commit
```



EXEMPLE

```
[alias]
pl          = pull --ff-only
pr          = pull --rebase
l           = log --graph --date=format:%d-%m-%y | %H.%M
              --pretty=format:'%C(yellow)%h%Creset%C(cyan)
              %cd%Creset %s %Creset%C(green)%an'
unstage     = reset -q HEAD --
discard     = checkout --
uncommit    = reset --mixed HEAD~
amend       = commit --amend
branches   = branch -a
stashes    = stash list
r           = checkout HEAD --
list        = diff-tree --no-commit-id --name-only -r
fp          = fetch --prune --all

[core]
excludesfile = ~/.gitignore
pager        = more -f -25

[user]
email        = training@zenika.com
name         = Zenika

[help]
autocorrect = 1
```





Lab 8

IGNORER DES FICHIERS

- Certains fichiers ne sont peut-être pas à commiter
- Ce genre de cas est géré par les fichiers `.gitignore`
- Le contenu d'un fichier `.gitignore` est une liste de fichiers/patterns à ignorer lors de l'utilisation de Git
- Quelques règles sont à connaître :
 - `#` Permet de placer un commentaire
 - `*` Remplace une suite de lettres dans le nom du fichier
 - `?` Remplace une lettre dans le nom du fichier
 - `!` Permet d'annuler un ignore fait précédemment
 - `/` Au début du nom d'une ressource permet d'ignorer cette ressource uniquement depuis le dossier où se situe le `.gitignore`
 - `/` À la fin d'une ressource permet de ne s'appliquer qu'aux dossiers



LES HOOKS

- Il est possible d'imposer des règles à différents moments du cycle sous la forme de scripts exécutables
- Les hooks sont disponibles dans `.git/hooks`
- On différencie les hooks côté client :
 - `pre-commit` : déclenché avant le commit, permet de lancer des checkstyles
 - `commit-msg` : déclenché lors du commit, permet de vérifier le format du message de commit
- Des hooks côté serveur :
 - `update` : exécuté lors des push utilisateurs
 - `acl` : gère les autorisations des utilisateurs



EXEMPLE DE HOOK

Exemple de vérification de format : fichier `.git/hooks/update`

```
$regex = /\[ref: (\d+)\]/

# vérification du format des messages de validation
def verif_format_message
  revs_manquees = `git rev-list #{$anciennerev}..#{$nouvellerev}`.split("\n")
  revs_manquees.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^\$/d'`
    if !$regex.match(message)
      puts "Le message de validation n'est pas conforme"
      exit 1
    end
  end
end

verif_format_message
```



INTÉGRATION SVN

- Git est utilisable comme client valide d'un serveur SVN avec les commandes de type `git svn`
- Avantages
 - Permet de profiter localement des avantages de Git
 - Passage progressif au fonctionnement de Git
- Désavantages
 - Version tronquée de Git
 - Préférable de garder une structure linéaire



COMPLÉTION/COULEURS/ÉDITEUR

- Un fichier de compléction bash `git-completion.bash` est fourni dans l'installation de Git et permet compléter les commandes grâce à la touche `tab`, ZSH supporte la compléction par défaut
- La gestion des couleurs dans le terminal s'active grâce à la configuration de `color.ui`
- Il est possible de personnaliser l'éditeur utilisé par Git lors des commits via `core.editor`



OUTILS GRAPHIQUES

- Une fois les concepts de Git bien maîtrisé il est possible d'utiliser des outils graphiques
- *Les outils graphiques cachent trop souvent ce qu'il se passe en dessous*
- Sourcetree ou GitKraken sont des outils modernes et puissants



GIT CHEAT SHEET

Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

Observe your Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

Working with Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new_branch

```
$ git branch new_branch
```

Delete the branch called my_branch

```
$ git branch -d my_branch
```

Merge branch_a into branch_b

```
$ git checkout branch_b
```

```
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```

Make a change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

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

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

```
$ git reset --hard
```

Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

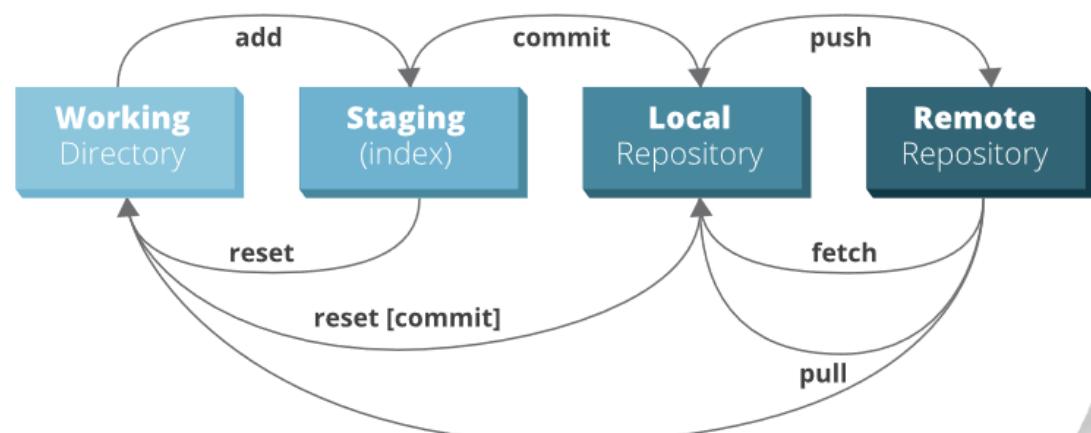
```
$ git push
```

Finally!

When in doubt, use git help

```
$ git command --help
```

Or visit <https://training.github.com/> for official GitHub training.



Source: <https://www.jrebel.com/blog/git-cheat-sheet>

GITLAB, BITBUCKET, GITHUB, ...

- GitLab, Bitbucket, GitHub
- **Facilite l'administration** d'un ensemble de dépôts Git partagés
- Simplifie la **gestion** des utilisateurs
- Permet d'imposer une politique de branche / convention de nommage
- Permet de gérer les sauvegardes
- Permet de gérer les **pull-requests**
- Facilite la mise en place de relectures de code



QUESTIONS





METTRE EN PLACE DES ORGANISATIONS DE TRAVAIL



PLAN

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes
- *Mettre en place des organisations de travail*
- Conclusion
- Projets OpenSource & Pull Requests
- Allons plus loin





DES ORGANISATIONS DE TRAVAIL

- Git est entièrement modulaire
- Git n'impose pas de cadre d'utilisation
- Facilité de faire et d'utiliser des branches
- Les DVCS permettent de dégager plusieurs patterns
 - Pas impossible avec un CVCS mais beaucoup plus coûteux



DES ORGANISATIONS DE TRAVAIL

- Questions :

- Quelle stratégie de versionning avez-vous adoptée ?
- Comment utilisez-vous les branches ?
- Avez-vous une organisation ?
- Tout le monde en est content ?





DES ORGANISATIONS DE TRAVAIL

- Plusieurs types d'organisations :
 - Organisation personnelle
Correspond à la manière dont le développeur travaille sur son poste local
 - Organisation de projet
Correspond à la manière dont les membres du projet partage leur code
 - Organisation globale
Correspond à la manière dont les différents projets s'organisent entre eux





DES ORGANISATIONS DE TRAVAIL

- 2 types d'entrepôts :

- Entrepôts publiques

Serveur Git depuis lequel les développeurs récupèrent les sources officielles

- Entrepôts de développeur

Serveur Git permettant à un développeur de publier son travail et de le rendre disponible



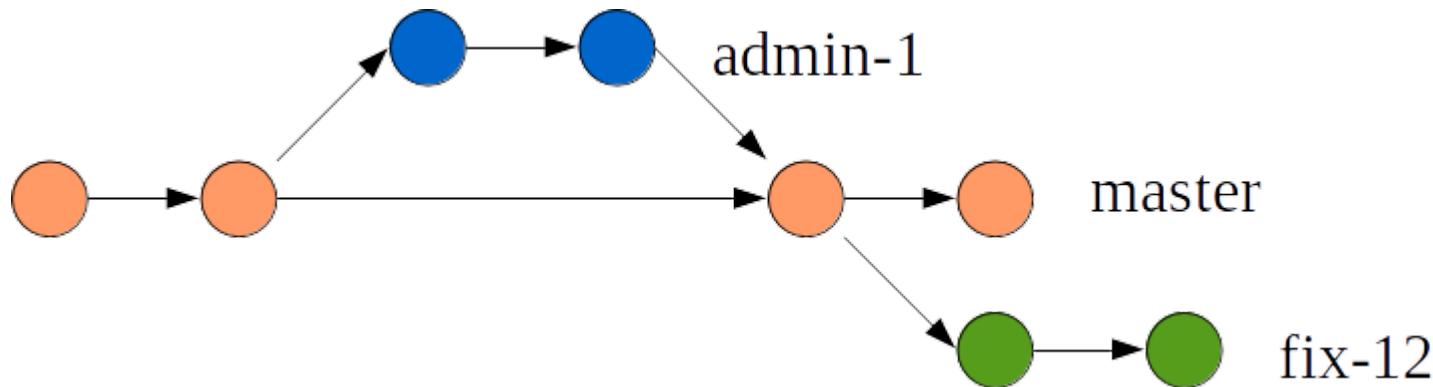
ORGANISATION PERSONNELLE

- Développement dans le tronc
- Utilisation "basique"



ORGANISATION PERSONNELLE

- Branche par fonctionnalité
- **master** = branche d'intégration
- Publier les branches de développement si elles sont reprises par d'autres
- Fusion des branches quand les fonctionnalités sont terminées





LES ORGANISATIONS DE PROJETS

- L'utilisation et la facilité de créer des branches permet de pousser les scénarios plus loin qu'avec les CVCS
- Modélisation de la chaîne de vie du projet
- La topologie reste généralement la même mais la symbolique change



LES ORGANISATIONS DE PROJETS

Il existe plusieurs éléments à prendre en compte lorsqu'on définit un workflow avec Git :

- le contenu des dépôts
- la gestion des branches et des tags
- la gestion du workflow d'équipe (CI/CD, intégration, production, ...)





LE CONTENU DES RÉPERTOIRES

- Un dépôt Git = un module applicatif (lib, jar, war, exe, dll, ...)
- Critères : tous les livrables d'un dépôt Git doivent avoir le même cycle de vie, release, ...
- Éviter les objets binaires à l'intérieur d'un dépôt Git (jar, doc/ppt, ...) (.gitignore + hooks)



LA GESTION DES BRANCHES

- Les **branches** ont une vocation **temporaire**, elles peuvent être déplacées ou même disparaître
- Les **tags** ont une vocation **permanente**, ils visent à garder un **état donné**, reproductible, et ne peuvent être ni modifiés ni supprimés

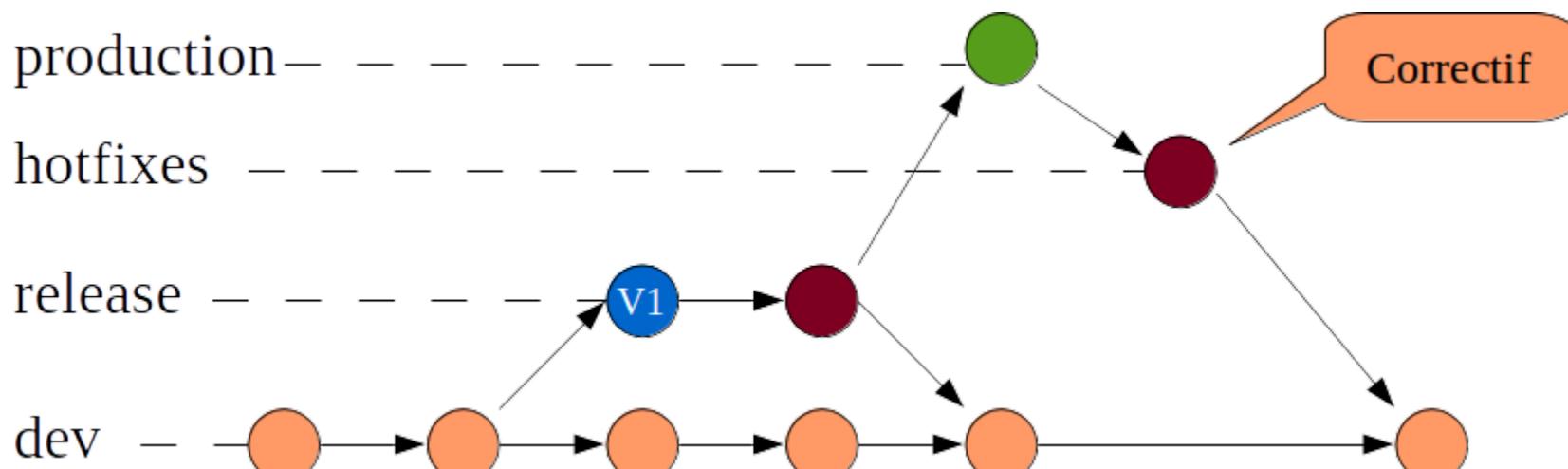
Quand on définit un workflow il faut :

- simplifier le travail des équipes
- produire un historique lisible et **exploitable**
- favoriser les intégrations régulières



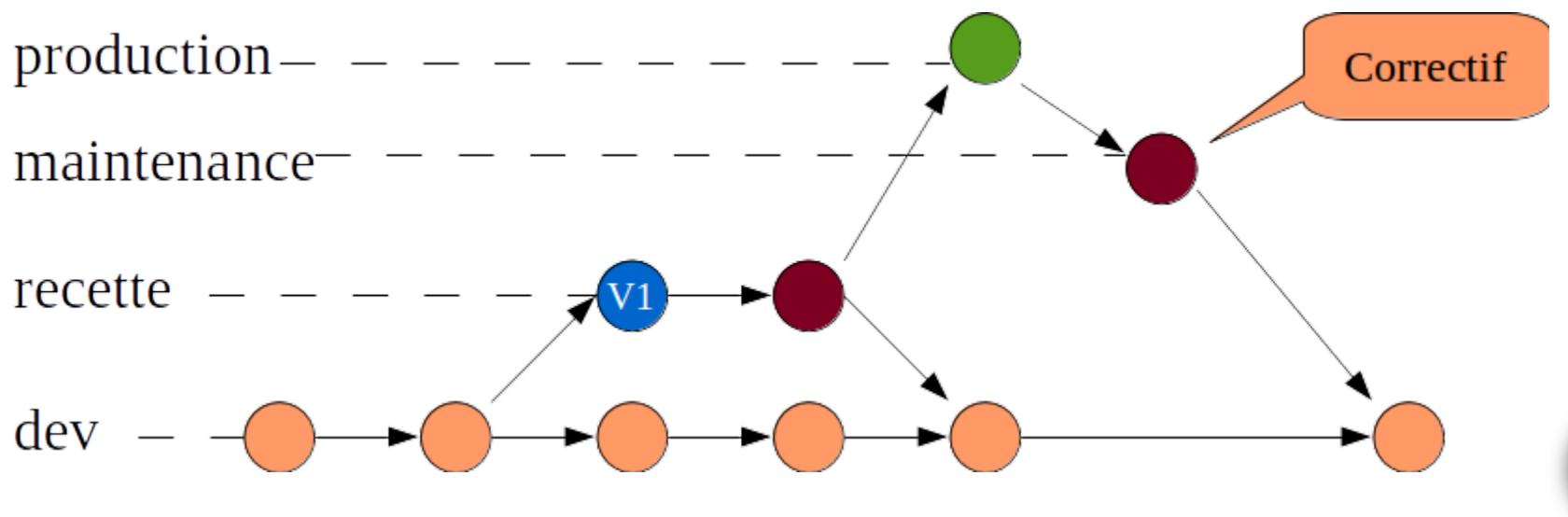
LES ORGANISATIONS DE PROJETS

- Des branches par étapes :
 - Branche d'intégration
 - Branche de livraison
 - Branche de maintenance



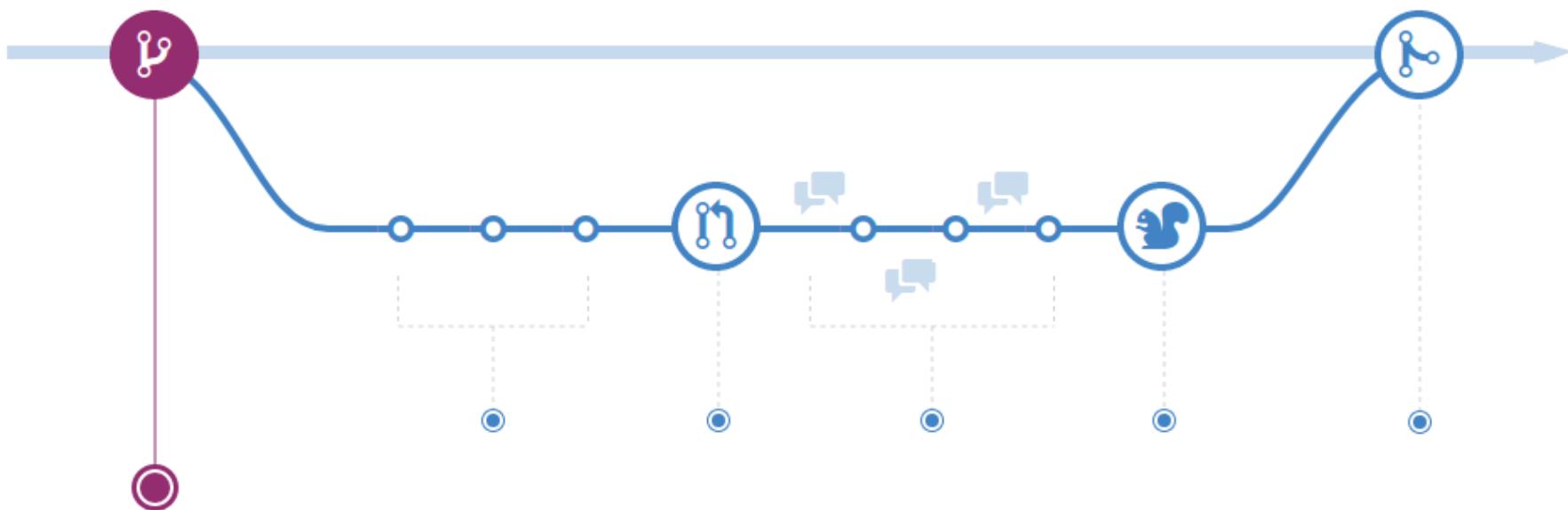
LES ORGANISATIONS DE PROJETS

- Des branches par environnements
 - Branche de production
 - Branche de recette
 - Branche de développement
 - Branche de site utilisateur



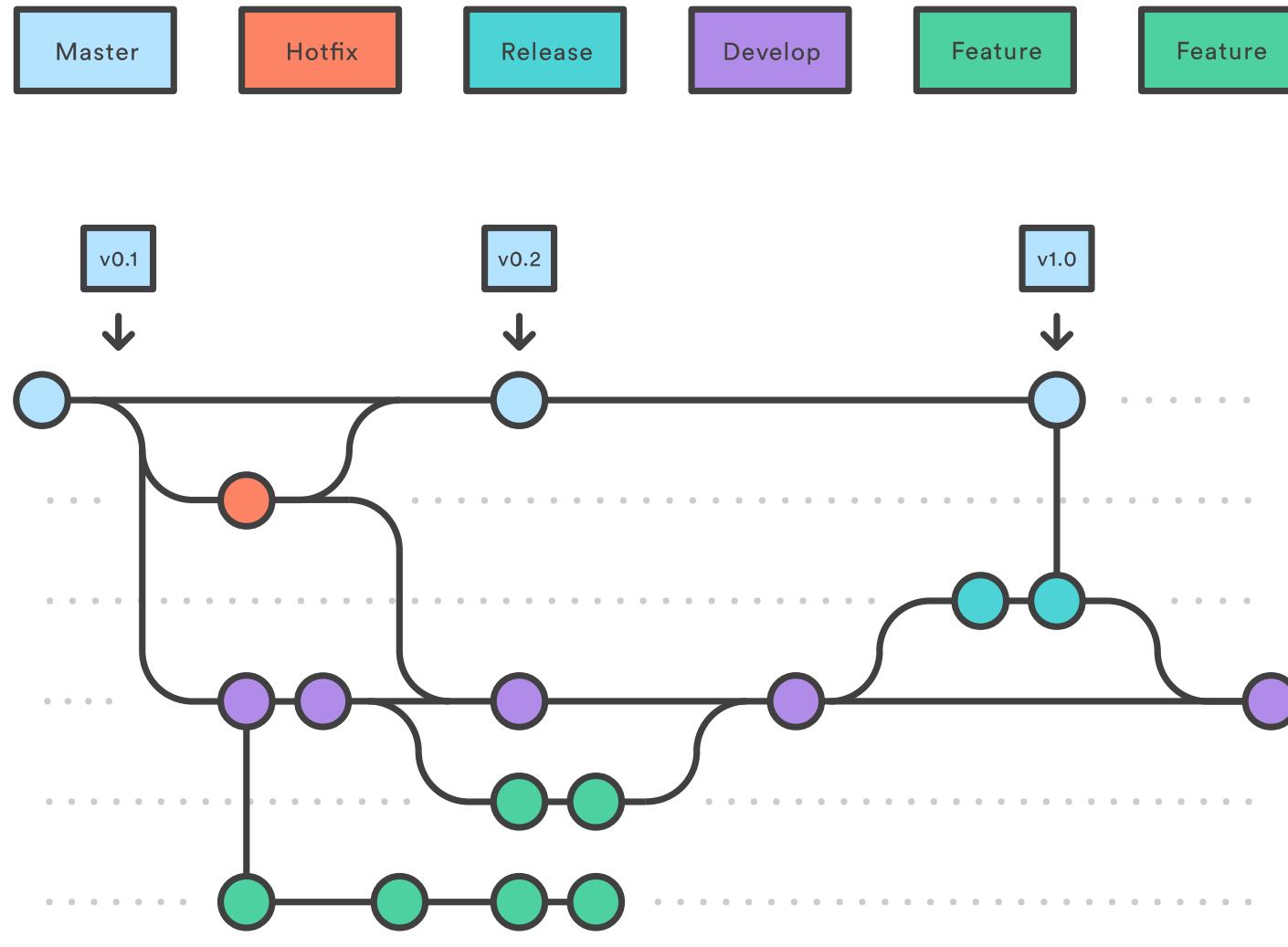
LES ORGANISATIONS DE PROJETS

- Le **Github Flow**
 - Branche principale
 - Branches de travail
 - Pull Request



LES ORGANISATIONS DE PROJETS

- Le **GitFlow**





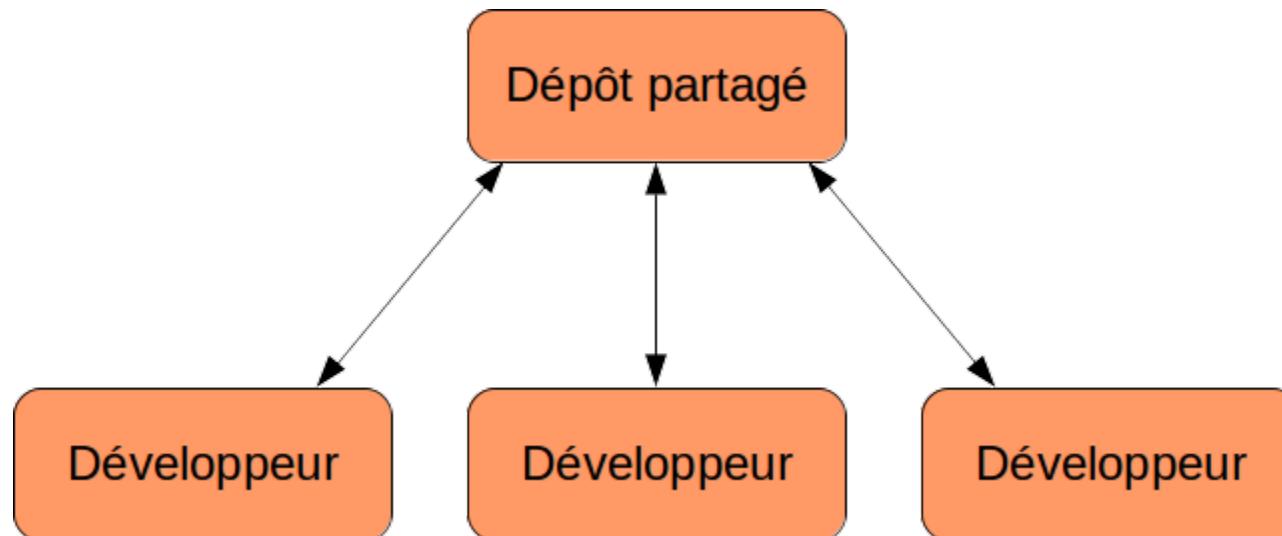
LES ORGANISATIONS DE PROJETS

- Le **GitFlow**
 - Branche de production
 - Branche de développement
 - Branches de travail
 - Branches de correctifs
 - Branches de releases



ORGANISATION GLOBALE

- Organisation "centralisée"



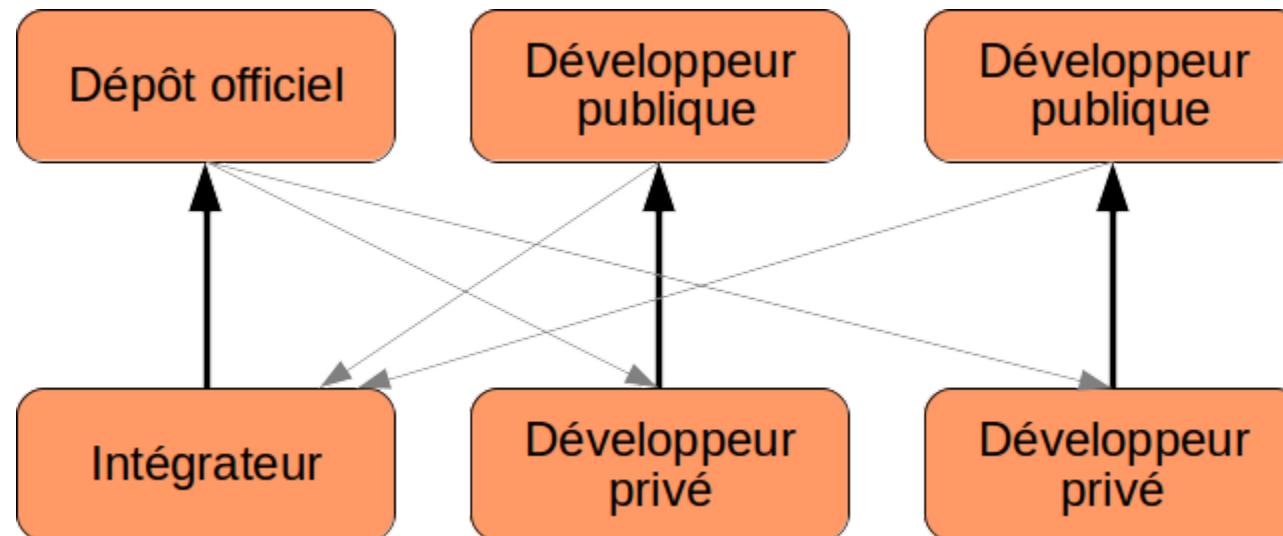
LE SERVEUR CENTRAL

- Avantages :
 - Les développeurs publient sur un dépôt partagé
 - Facilité pour les migrations depuis un outil de gestion de version centralisé comme SVN
 - Partage rapide des sources
- Inconvénients :
 - Chaque développeur doit intégrer son travail à celui des autres avant de publier
 - Si toutes les branches sont partagées, l'historique n'est pas lisible



NIVEAU D'ORGANISATION

- Organisation avec intégrateur



L'INTÉGRATEUR

- Avantages :

- Les développeurs ont chacun des dépôts personnels publics
- Un intégrateur rassemble le travail pour créer les versions de livraison dans un dépôt public officiel
- L'historique peut être nettoyé, ordonné et publié de manière centralisé

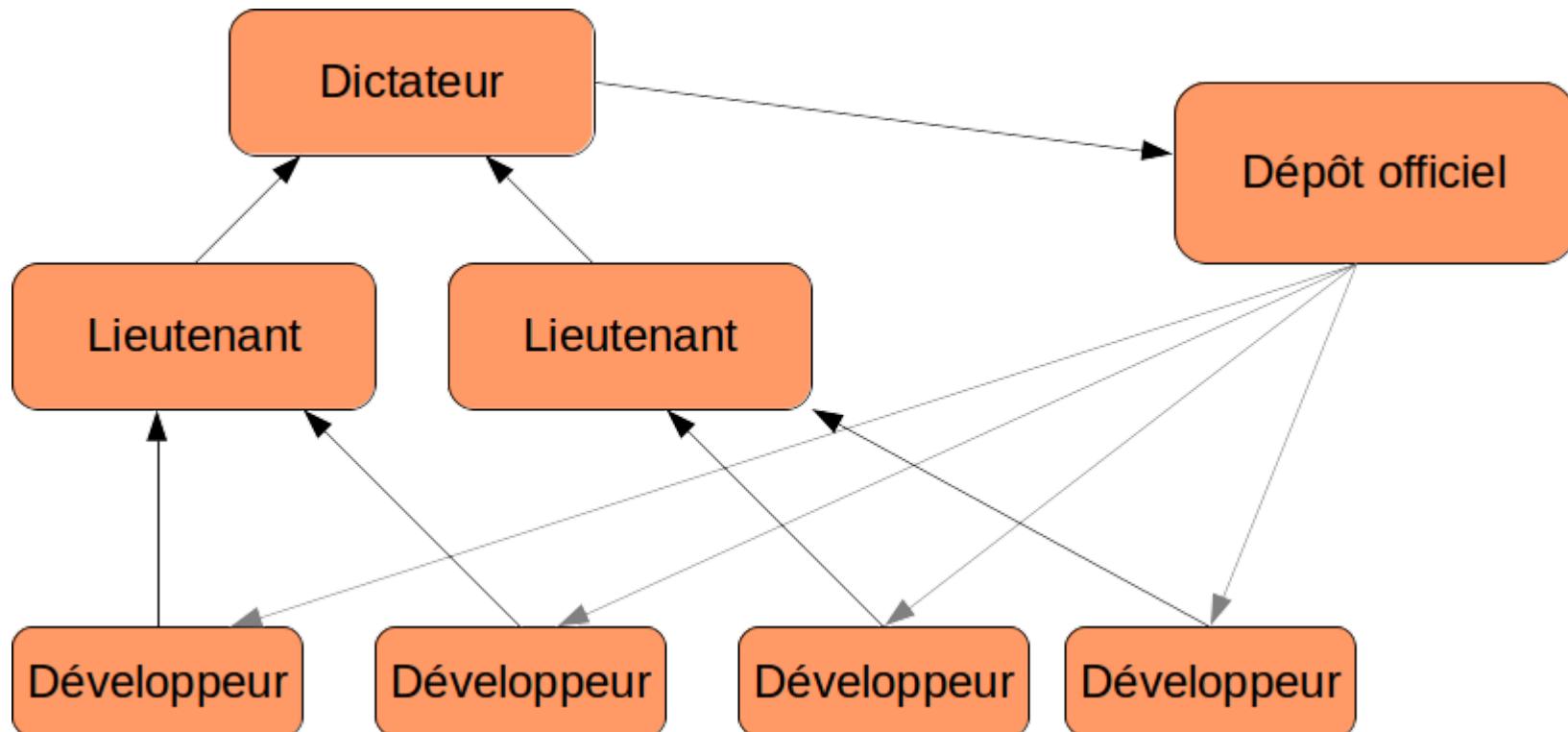
- Inconvénients :

- Le partage des sources de chaque développeur peut être long, prévoir des cycles de développements courts
- Ne pas intégrer à la dernière minute



NIVEAU D'ORGANISATION

- Organisation du "dictateur bienveillant"



LE "DICTATEUR BIENVEILLANT"

- Avantages :
 - Modèle du noyau Linux
 - Le modèle pyramidal est utile pour les projets multi-équipes
- Inconvénients :
 - Nécessite beaucoup d'organisation
 - Les releases sont minutieusement préparées







RAPPELS & CONCLUSION



PLAN

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes
- Mettre en place des organisations de travail
- *Conclusion*
- Projets OpenSource & Pull Requests
- Allons plus loin



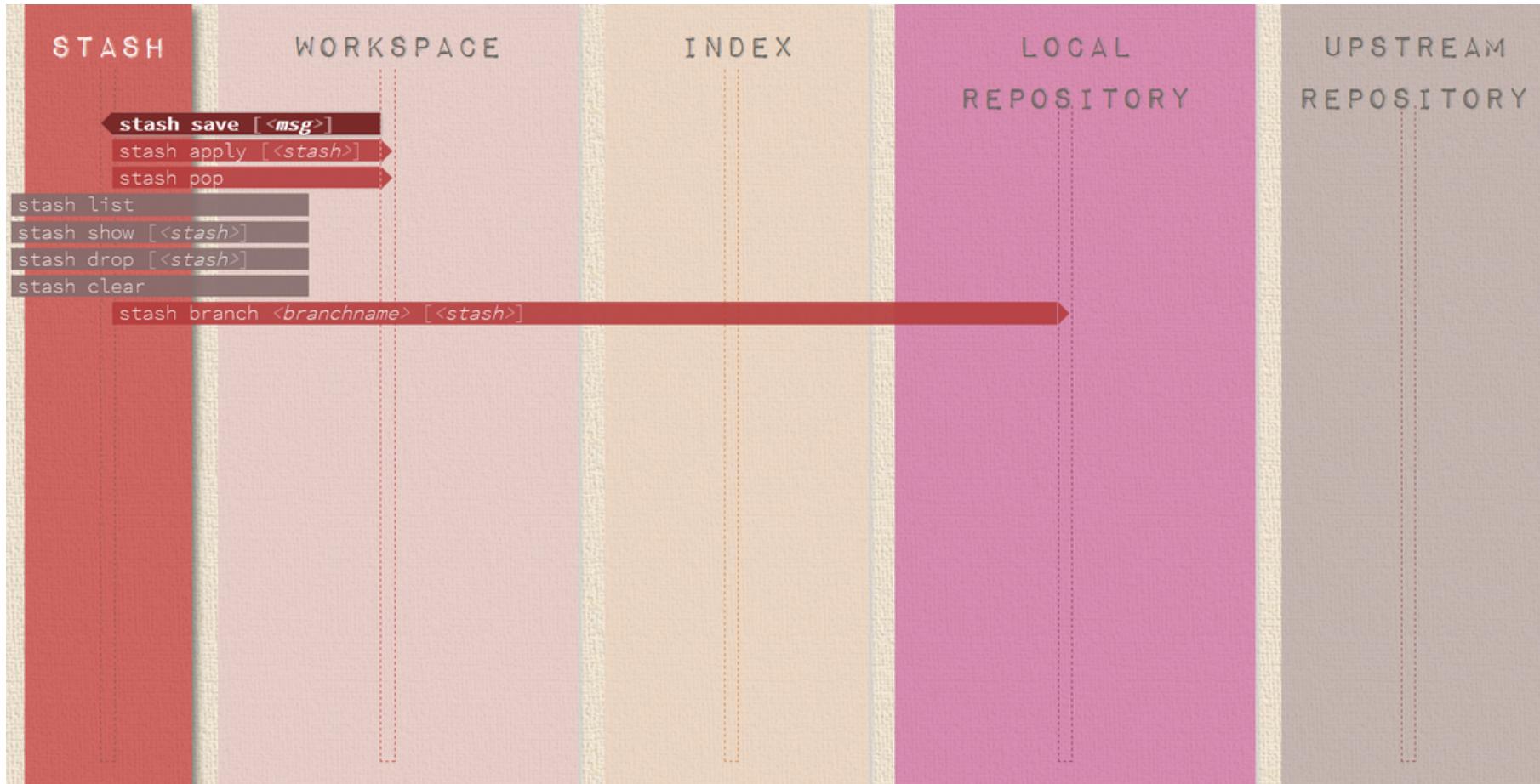
RAPPELS

Rappel des différentes zones dans Git et des commandes d'interaction avec ces zones.

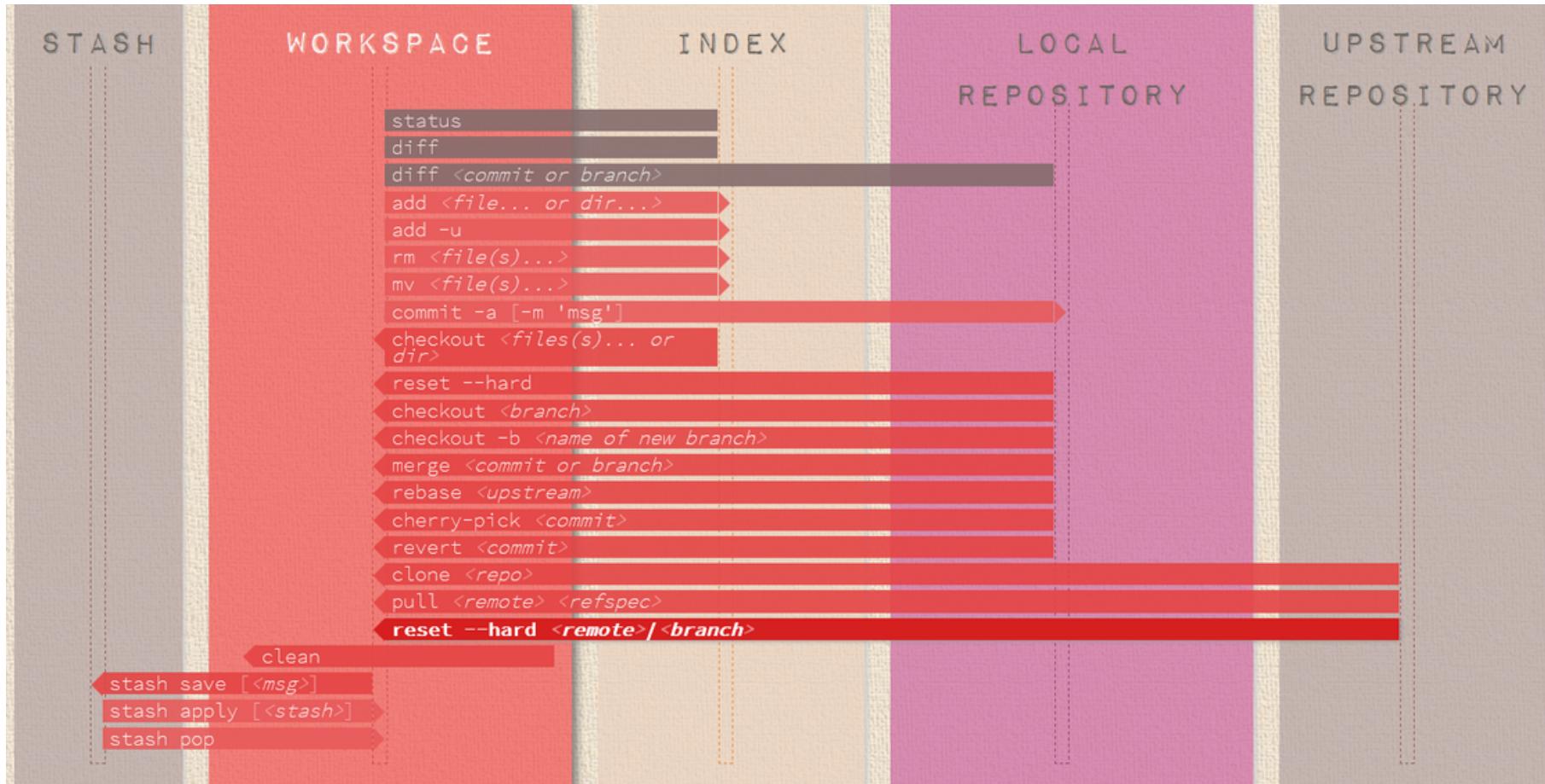
Issue du [cheat sheet de NDP Software](#).



STASH



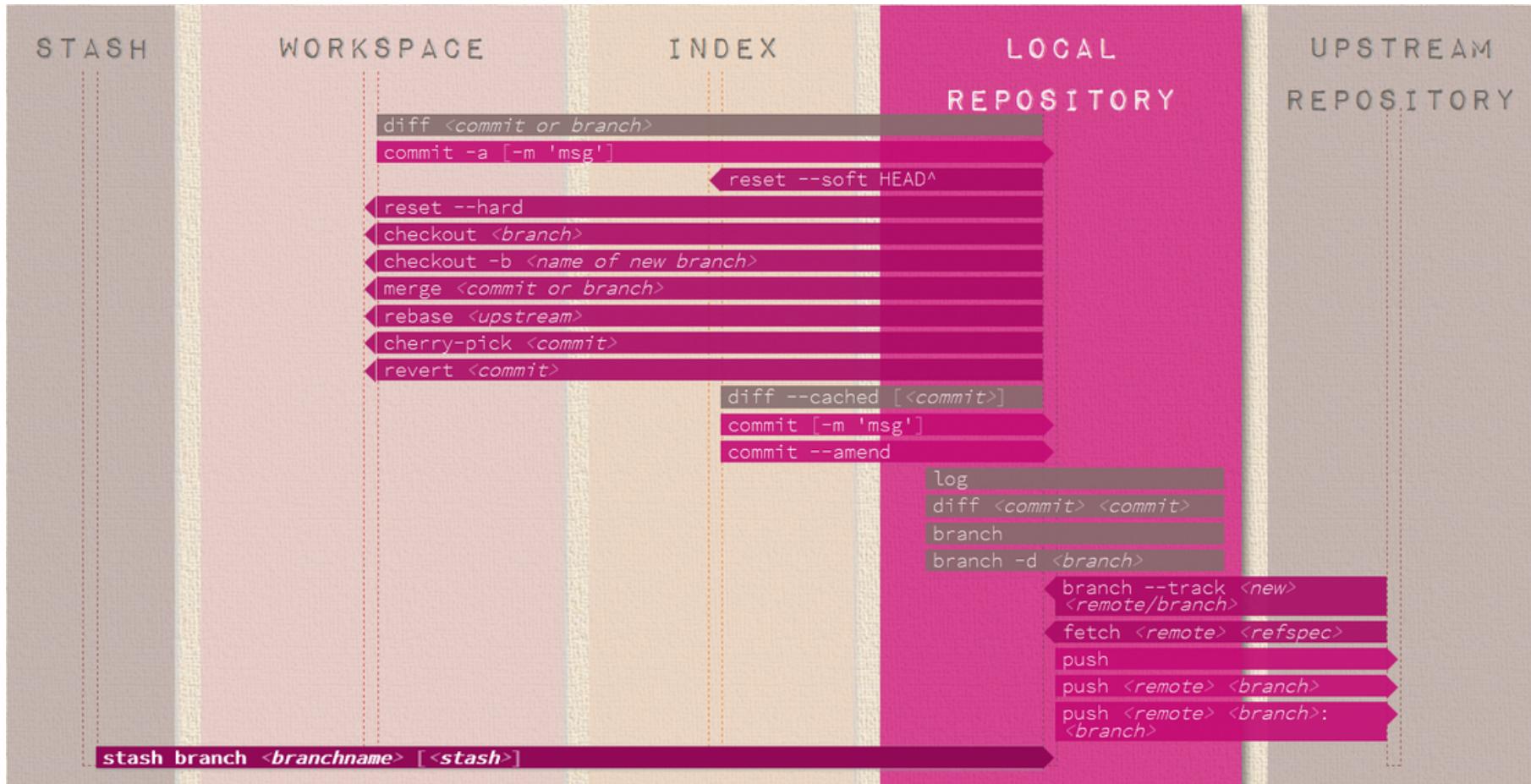
WORKING DIRECTORY



INDEX



DÉPÔT LOCAL



DÉPÔT DISTANT



CONCLUSION

- Git est un **outil** puissant, qui simplifie la collaboration
- Vous devez le maîtriser pour pleinement l'utiliser



QUESTIONS





PROJETS
OPENSOURCE & PULL
REQUESTS



PLAN

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes
- Mettre en place des organisations de travail
- Conclusion
- *Projets OpenSource & Pull Requests*
- Allons plus loin



PROJET OPEN SOURCE

- Logiciel dont le code source est mis à disposition gratuitement
- Utilisation d'une licence validée par la FSF (Free Software Foundation)
 - :
 - GPL, LGPL
 - ASF, BSD, MIT, EPL
 - Si pas de licence présente : all rights reserved (mais dépend du site hébergeur)
- Mise à disposition de documentation
 - Pour les utilisateurs
 - Pour les développeurs



PROJET OPEN SOURCE

- Afin de faciliter les contributions externes :
 - Avoir un projet facile à builder (Maven, Gradle, Makefile, etc.)
 - Avoir un projet facile à démarrer (pour pouvoir tester rapidement une modification locale)
 - Avoir un bugtracker
- Fournir des règles à respecter :
 - Style de code / Contenu d'un message de commit
 - Présence de tests correspondants à une fonctionnalité
- Éventuellement :
 - Signature d'un CLA (Contributor Licence Agreement)



CONTRIBUTIONS EXTERNES

- Sans DVCS :
 - Accès direct au dépôt : réservé aux contributeurs officiels
 - Envoi de patchs au format **diff**, en PJ d'une fiche de bugtracker, par email
- Avec un DVCS :
 - Notion de pull request : mise en avant par GitHub, reprise par l'ensemble des autres outils (merge request, etc...)
 - Pull request :
 - "à la main" : mise à disposition d'une branche sur un repo personnel public, ajout d'un remote, merge manuel
 - "avec un outil" : utilisation des assistants fournis, fils de discussion, commentaires au sein du code, etc



PULL REQUEST GITHUB

- Workflow :

- Fork GitHub du projet par l'utilisateur 
- Clone du projet
- Création d'une branche locale, modifications, commits
- Push de la branche locale sur le fork GitHub
- Création d'une pull request de la branche
 - Du fork vers le projet principal 
- Discussion avec un contributeur du projet
- Ajout éventuel de commits / corrections
- Merge
- Fermeture de la pull request







ALLONS PLUS LOIN



PLAN

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes
- Mettre en place des organisations de travail
- Conclusion
- Projets OpenSource & Pull Requests
- *Allons plus loin*



GIT HOOKS

Les **hooks** Git sont des scripts que **git** exécute avant ou après certains événements comme : **commit**, **push** ou **receive**. Les hooks sont une fonctionnalité **inclus**e dans git, il n'y a pas besoin d'installer de logiciel supplémentaire. Les hooks sont exécutés localement.

Ces hooks sont des scripts, ils ne sont donc limités que par l'imagination des développeurs. Voici quelques exemples :

- ***pre-commit***: Vérification des fautes, et des messages associés
- ***pre-receive***: Vérification des règles de codage.
- ***post-commit***: Envoie d'Email/SMS aux membres de l'équipe.
- ***post-receive***: Déploiement du code en production.

Les hooks sont exécutés **server side**, mais ils peuvent (et doivent) être copiés **client side** (dans `.git/hooks`).



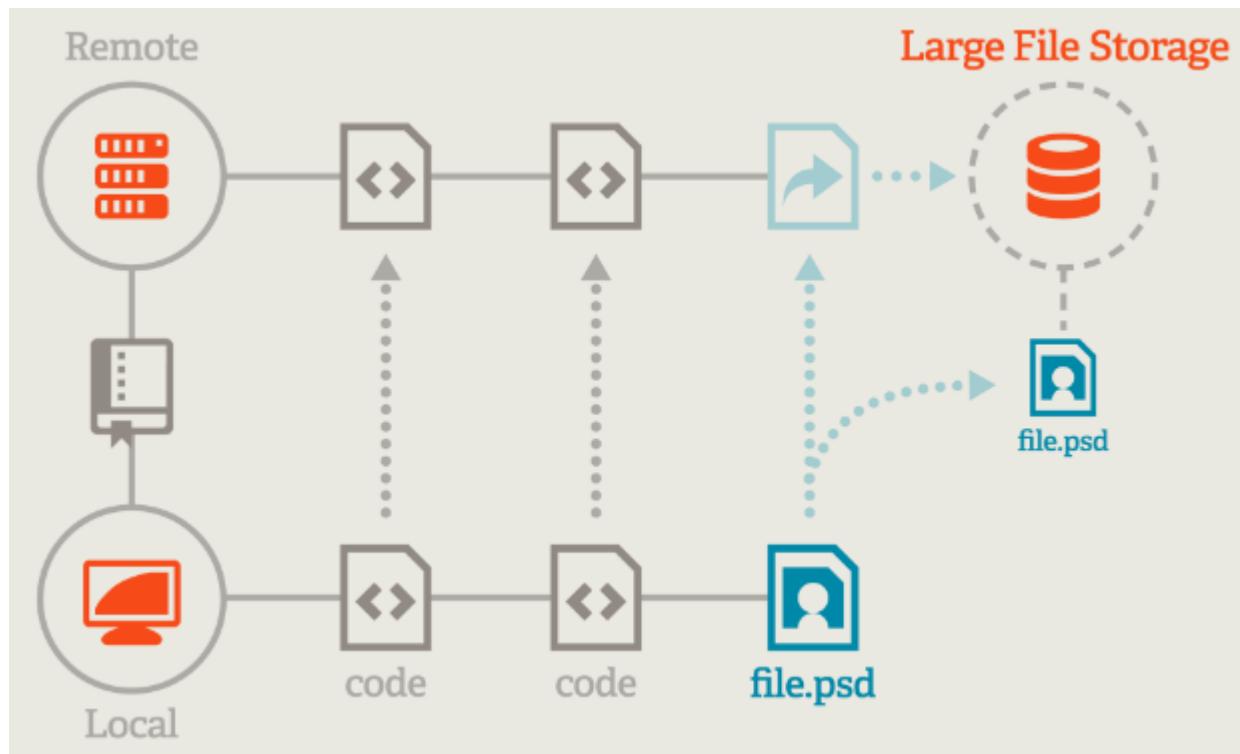
GIT LOG VS GIT REFLOG

- Git log montre **l'état courant** d'un dépôt
- Reference logs, ou "**reflogs**", enregistre les actions modifiant les références localement.
- Les Reflogs sont utiles dans plusieurs commandes Git pour spécifier les **anciennes valeur** d'une références.
- Par exemple, **HEAD@{2}** signifie "l'ancienne valeur de HEAD utilisée deux commandes en arrière", **master@{one.week.ago}** signifie "la position de master il y a une semaine dans ce dépôt local".
- exemple : **git reflog master@{one.week.ago}**



GIT LFS

- **Git Large File Storage (LFS)** remplace les fichiers volumineux tels que: vidéos, datasets, images par des pointeurs. Il stocke le contenu sur un autre serveur distant.



POUR UTILISER L'EXTENSION LFS

- Télécharger et installer l'extension. Il ne faut le faire qu'une fois.
| git lfs install
- Définissez les types de fichiers que vous voulez gérer avec Git LFS (ou modifier votre .gitattributes). La liste peut être modifiée à tout moment.
| git lfs track "*.psd"
- Vérifiez que le fichier .gitattributes est sous Git
| git add .gitattributes
- Et c'est fini. Il suffit de pousser comme avant. La configuration **server side** est faite par votre gentil admin.
| git add file.psd
| git commit -m "Add design file"
| git push origin master



PIÈGE COURANT

- Une partie des utilisateurs avec LFS non configuré.
Ces utilisateurs n'auront que des pointeurs, et ils ne seront pas capables de voir/lire/modifier ces fichiers.



CONVERTIR UN DÉPÔT AVEC DES FICHIERS LFS

- Créer un clone "bare" du dépôt.

```
git clone --bare https://myurl.com/old-repository.git
```

- Aller dans le dépôt cloné.

```
cd old-repository.git
```

- Récupérer les objets LFS du dépôt.

```
git lfs fetch --all
```

- Publier en mode miroir vers le nouveau dépôt.

```
git push --mirror https://myurl.com/new-repository.git
```

- Publier les objets LFS vers le miroir.

```
git lfs push --all https://myurl.com/new-repository.git
```

- Supprimer le dépôt local créé en première étape.

```
cd .. ; rm -rf old-repository.git
```



GIT WORKTREE

La **situation** : on ne travaille pas qu'avec des fichiers sources, mais aussi avec des fichiers temporaires (jar, dll, ...). Il peut être long de collecter/générer ces fichiers.

Une solution est **git worktree** : permet de gérer plusieurs espaces de travail liés au même dépôt.

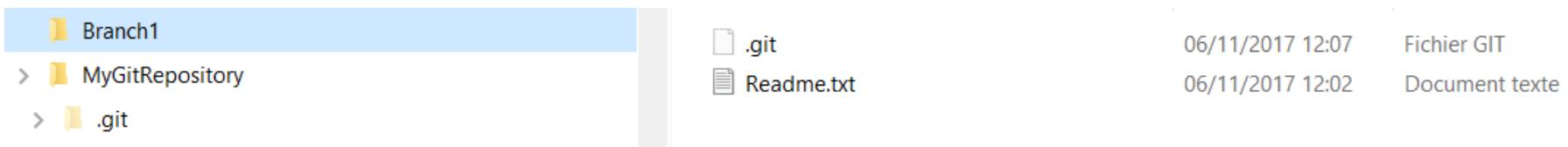
Un dépôt Git peut avoir plusieurs espaces de travail (**working area**) vous permettant de travailler dans plusieurs branches en parallèle. Cet espace est un **linked working tree** par opposition au **main working tree** créé par **git init** ou **git clone**.



GIT WORKTREE

L'organisation des répertoires est simple:

- votre répertoire principal créé par le **clone**
- un répertoire par branche ou tag sur lequel vous travaillez
- Vous pouvez ajouter ou supprimer autant d'espaces de travail que vous désirez



La magie est faite par le fichier **.git** à l'intérieur du répertoire.

```
$ cat Branch1/.git  
gitdir: ../../MyGitRepository/.git/worktrees/test
```



GIT BISECT

Cette commande utilise un algorithme de recherche dichotomique pour retrouver un commit particulier dans votre historique.

Vous devez définir un **mauvais** commit, connu pour avoir un bug, et un **bon** commit avant que le bug ne soit introduit.

Alors, `git bisect` choisit un commit entre ces deux points et vous demande si le commit est **bon** ou **mauvais**.

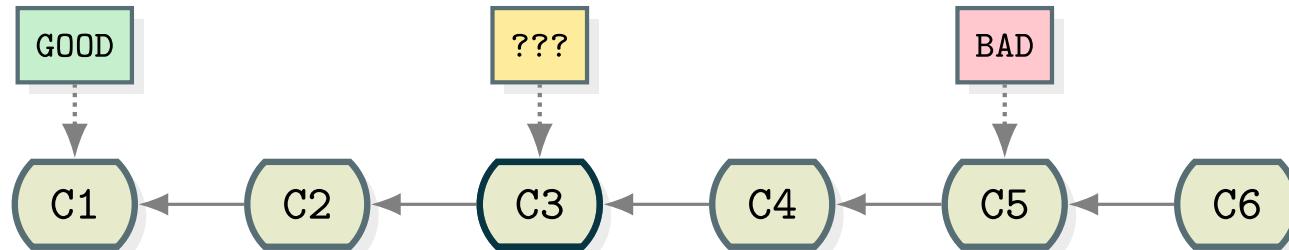
Il continue alors par itération jusqu'à trouver le commit ayant introduit le changement.

`git bisect` peut être utilisé pour trouver/identifier tout changement d'une propriété de votre projet.

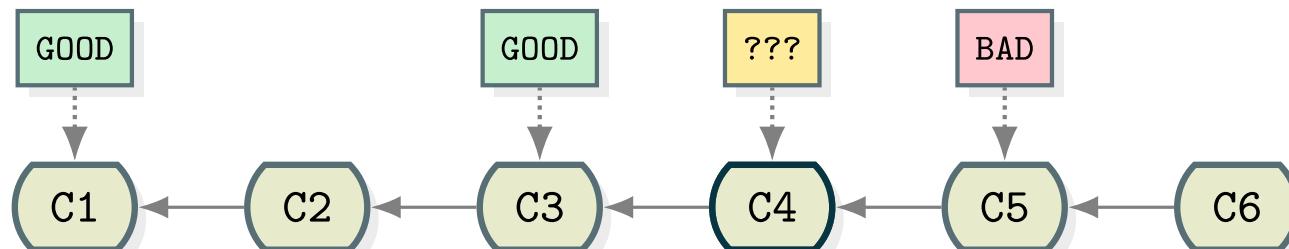


GIT BISECT EN ACTION

`git bisect` nécessite un **GOOD** et un **BAD** commit pour démarrer.



Le coût est en $\log(n)$. Même sur un historique important c'est efficace.
Pensez à l'automatiser.



GIT RESET

- `git reset` ne se limite pas à **une** opération.
- **Unstage** / invalider un contenu
- **Modifier** un travail **en cours**
 - Annuler un ajout en vue d'un futur commit
 - Supprimer des modifications en cours dans notre WD
- **Modifier** le **dernier commit**
 - Modifier le dernier commit (retirer une partie)
 - Ah mince, j'ai oublié certaines modifications dans mon dernier commit



GIT RESET : MAIS AUSSI

- **Les branches**
 - Finalement, mes X derniers commits auraient du être sur une branche
 - Le travail de mes X derniers commits n'est plus pertinent
 - Je souhaite annuler mon dernier **merge**, **pull** ou **rebase**
- Euh.... **j'ai fait n'importe quoi** avec **git reset**
- L'option **--keep**



UNSTAGE / INVALIDER UN CONTENU

1/3

Git nous fournit des informations précieuses via `git status`.

Créons un projet intitulé ***git-reset-example*** et ajoutons-lui deux fichiers ***README.md*** et ***.gitignore*** :

```
mkdir git-reset-example
cd git-reset-example
git init
echo 'TODO' > README.md
echo 'tmp' > .gitignore
```

Pour le moment aucun fichier n'est versionné / géré par Git.

```
$ git status

On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```



UNSTAGE / INVALIDER UN CONTENU

2/3

Préparons donc notre premier commit:

```
$git add README.md .gitignore
```

```
$ git status
```

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

```
new file: .gitignore
```

```
new file: README.md
```

Note : Il s'agit d'un cas particulier lié au tout premier commit sur un projet (autrement appelé **root commit**).

Nous n'avons pas accès à **git reset** mais **git rm --cached <file>...**



UNSTAGE / INVALIDER UN CONTENU

3/3

Finalisons désormais notre premier commit :

```
$ git commit -m "Initial commit"
```

Ajoutons à présent un nouveau fichier avec la date du jour :

```
$ date > date.txt  
$ git add date.txt
```

Cette fois-ci **git status** nous indique bien que pour invalider cet ajout à l'index, il nous suffirait de faire un **git reset HEAD date.txt** (en l'espèce on pourrait même omettre le **HEAD**).

```
$ git status  
  
On branch master  
Changes to be committed:  
  (use "git reset HEAD <file>..." to unstage)  
  
    new file:   date.txt
```



ANNULER UN AJOUT EN VUE D'UN FUTUR COMMIT

- Cas : on veut retirer certains fichiers du commit suivant.
 - ex : `git add *` ... oups j'ai ajouté ***private.sh***

```
| $ git reset private.sh
```



SUPPRIMER DES MODIFICATIONS EN COURS DANS NOTRE WD

Pour cela il vous suffit d'appliquer un `git reset --hard HEAD`.

Attention : Notez bien que dans ce cas de figure vos travaux n'ont pas été versionnés/historisés et donc vous les perdrez définitivement !

Reprendons par exemple notre fichier `date.txt` :

```
| date > date.txt  
| git reset --hard  
| date >> date.txt
```

La commande `git reset` (ligne 2) utilise **implicitement HEAD** si nous ne lui précisons pas. Dans ce cas précis, c'est un peu violent (`reset` de tout le working dir).

Nous avons cependant une alternative:

```
| git checkout HEAD date.txt
```

Il s'agit d'un **checkout** partiel, qui permet d'aller prendre la version désirée d'un fichier.



MODIFIER LE DERNIER COMMIT (RETIROUER UNE PARTIE)

- `git reset` avant le `commit`, mais que faire si on ne s'en aperçoit qu'après ?

Pas de panique, il **suffit de défaire le dernier commit et de retirer le fichier incriminé**, puis de **rejouer le commit**.

On a ici plusieurs approches, la manière **100% reset** consiste à:

- ramener le HEAD d'un cran en arrière tout en conservant l'état actuel dans le stage et le WD,
- retirer notre fichier (ou partie de fichier).

Revenons sur l'opération fautive...

```
git add .
git commit -m "Ajout de la clé publique"
```



LA RÉSOLUTION...

```
git reset --soft HEAD~1      #nous ramenons le HEAD, et juste lui (ni le stage,  
# ni le WD) d'un cran en arrière dans notre  
# historique public (HEAD~1)  
  
git reset private.sh        # sort le private.sh du stage  
echo 'private.sh' >> .gitignore  
git add .gitignore  
git commit -m "Ajout de la clé publique" # on recommite tout proprement
```

En pratique, on aurait recours à **git commit --amend** dans la vraie vie pour ce type de besoin :

```
git reset HEAD~1 private.sh  
echo 'private.sh' >> .gitignore  
git add .gitignore  
git commit --amend --no-edit
```

En fait **git commit --amend --no-edit** est équivalent à **git reset --soft HEAD~1** suivi de **git commit -C ORIG_HEAD**.



J'AI OUBLIÉ CERTAINES MODIFICATIONS

Plusieurs cas de figure :

- Vous avez oublié d'ajouter certains fichiers.
- Vous souhaitez modifier le message de commit.

Ceci ressemble beaucoup à notre situation précédente : On veut revenir au stage précédent, effectuer une éventuelle mise à jour, puis recréer notre commit.

```
git reset --soft HEAD~1      # revenons à l'état précédent  
# ajouts de fichier          # effectuons la mise à jour  
git commit -m "..."          # on refait notre commit
```

Là aussi, **git commit --amend** nous fera gagner du temps :

```
# ajouts de fichier  
git commit --amend --no-edit
```

Si on souhaite modifier le message, il suffit de remplacer le **--no-edit** par un **-m "..."**.



GIT RESET ET LES BRANCHES

Mes X derniers commits auraient du être sur une branche

Vous n'avez pas encore les bons réflexes et vous avez travaillé sur **master** et effectué 3 commits.

Par chance vous vous en rendez compte avant de modifier le dépôt distant avec un **git push**.

On veut marquer nos derniers commits comme appartenant à la branche **features/f1**, et que **master** revienne de son côté **3 crans en arrière**.

Souvenez-vous : une branche est une étiquette.

```
git branch features/f1      #Calage de l'étiquette de branche  
git reset --soft HEAD~3    #Recalage de l'étiquette de branche master  
git checkout features/f1   #Activation de la nouvelle branche
```



LE TRAVAIL DE MES X DERNIERS COMMITS N'EST PLUS PERTINENT

Les 3 derniers commits ne sont plus pertinents...

On souhaite donc les retirer de notre historique.

```
| git reset --hard HEAD~3      #Suppression de 3 commits
```



ANNULER SON DERNIER MERGE, PULL OU REBASE

En gros vous avez effectué une opération modifiante importante que vous souhaitez finalement défaire :

- après un **merge de contrôle** (fusion permettant de vérifier qu'une branche s'intègre bien dans sa branche parente) ;
- après avoir **récupéré l'historique** mis à disposition par vos collègues (**pull**) et vous être rendu compte que vous n'étiez finalement pas prêt ;
- après avoir mis à jour votre branche « par-dessus » une autre (par exemple vous avez fait un **rebase** à l'envers).

Pas de panique, **git reset** va gérer ça tranquillement !

Souvenons nous de **HEAD** et de **ORIG_HEAD**, qui garde la trace de la dernière référence avant une opération majeure.

```
| git reset --merge ORIG_HEAD
```



AU SECOURS, J'AI FAIT N'IMPORTE QUOI AVEC GIT RESET

Il peut arriver que vous soyez **un peu perdu** après avoir voulu utiliser **git reset**.

Sachez qu'il n'y a qu'un cas de figure dans lequel votre travail sera perdu : lors de l'utilisation d'un **git reset --hard** alors que vous aviez des modifications non commitées dans votre WD.

Toute autre situation peut être démêlée grâce au journal des actions Git, le **reflog**.



GIT REFLOG

Ce journal nous fournit les positions successives de HEAD pour ce dépôt local.

```
$ git reflog -7
```

```
2154788 HEAD@{0}: reset: moving to HEAD~3
15cc480 HEAD@{1}: checkout: moving from features/f1 to master
3e221d0 HEAD@{2}: checkout: moving from master to features/f1
15cc480 HEAD@{3}: commit: Pseudo commit n°2
d17c8d6 HEAD@{4}: commit (amend): Pseudo commit n°1
d17c8d6 HEAD@{5}: commit: Pseudo commit
3e221d0 HEAD@{6}: commit: Ajout date
2154788 HEAD@{7}: commit (initial): Initial commit
```

La première colonne désigne les SHA-1 des commits référencés par HEAD à chaque étape.

La seconde colonne fournit la référence de journalisation (0 étant la dernière position en date).

Viennent ensuite les intitulés des actions concernées.



GIT REFLOG

Nous allons pouvoir utiliser ces références :

```
$ git reset --hard HEAD@{X}
```

Il y néanmoins un petit piège à éviter. L'emploi de `git reset --hard HEAD@{1}` va nous repositionner un cran en arrière. Il va également générer une nouvelle entrée dans le reflog et de ce fait, décaler la numérotation :

```
$ git reset --hard HEAD@{1}
$ git reflog -10
```

```
15cc480 HEAD@{0}: reset: moving to @{1}
2154788 HEAD@{1}: reset: moving to HEAD~3
15cc480 HEAD@{2}: checkout: moving from feat/f1 to master
...
```

Du coup, si on refait un `git reset --hard HEAD@{1}` on tourne en rond...



GIT REVERT

`git-revert` - Revert some existing commits

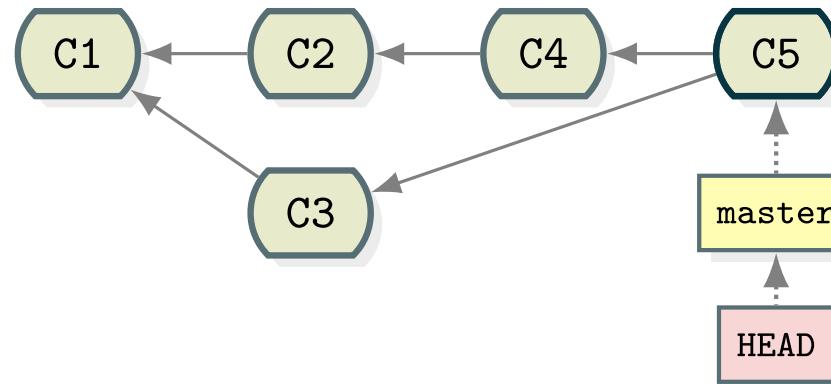
Rappel :

- Toutes les commandes Git sont des opérations sur le graphe
- Git garde une trace de toutes les actions



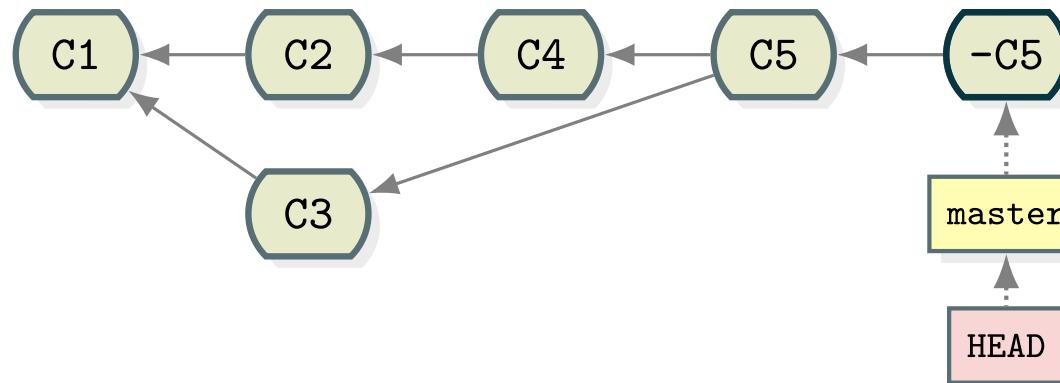
GIT REVERT EN ACTION

Positionnons nous juste après un **merge**.



GIT REVERT EN ACTION

Quand vous faites un **revert**, Git va faire un lot d'actions pour vous remettre dans le même état que précédemment.



Warning : Git n'effacera pas le noeud de merge, il remettra les fichiers dans un état souhaitable.



GIT REVERT TRAPS

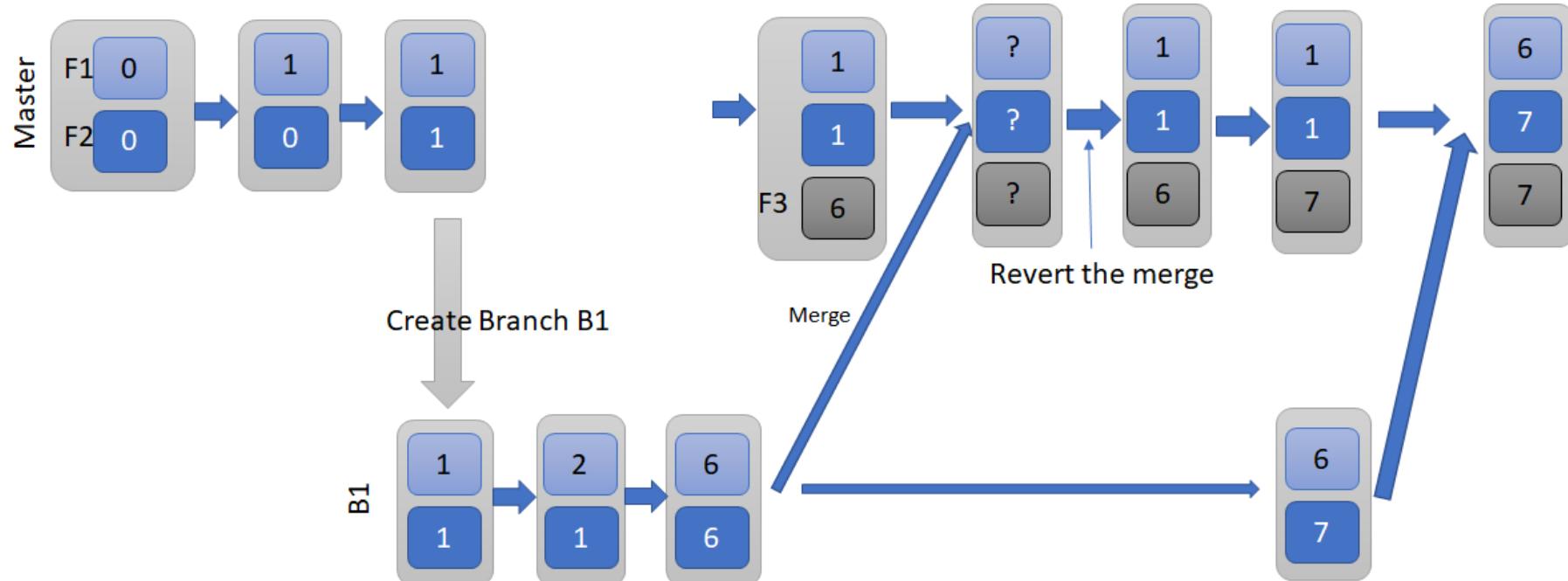
Scénario :

- on a une branche **master** dans laquelle travaille une équipe.
- un développeur fait une correction, et l'intègre avec la branche **master** (vendredi soir :))
- cela introduit une régression. Plutôt que de corriger, on choisit de faire un **revert** de son **merge**.
- à son retour lundi, il doit corriger son intégration et réintégrer son travail.

Informations : - **F1**, **F2**, **F3** sont 3 fichiers texte. Les nombres indiquent le contenu du fichier à un instant t. - Vous porterez une attention particulière à prédire le résultat des opérations de **merge**. - A vous de créer l'historique et de faire les différentes actions.



GIT REVERT TRAPS



QUESTIONS

