

# 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

# Logistique

- Horaires
- Déjeuner & pauses
- Autres questions ?



# Introduction

# Plan

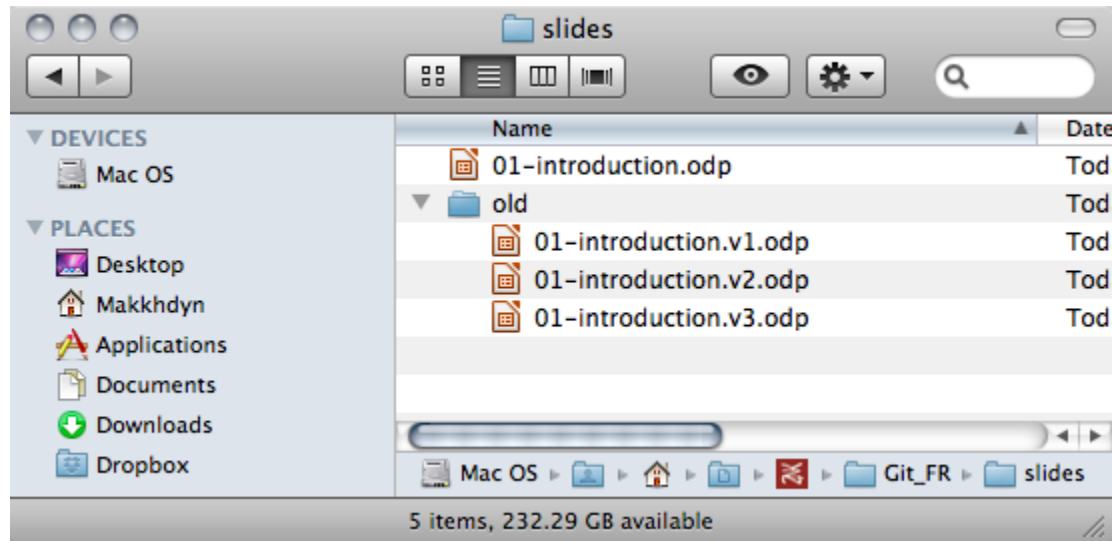
- *Introduction*
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes

# 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
  - Édition d'images
  - Bases de données
- Simplement toute activité résultant dans 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



- Puis s'améliore avec des systèmes plus adaptés





# 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



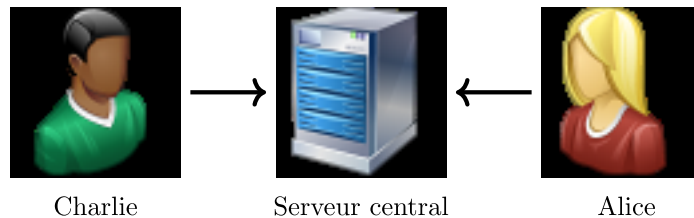
- Pour ça des systèmes de versioning plus élaborés ont été adoptés
- Regroupés sous le terme de V.C.S pour **V**ersion **C**ontrol **S**ystem
- Plusieurs fonctionnements existent selon les solutions choisies
  - Le fonctionnement traditionnel **centralisé** ou CVCS
  - Le fonctionnement **distribué** 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 - à 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 code doit compiler et les tests unitaires doivent passer
  - Le message associé doit indiquer le contenu des modifications
  - Il est trop souvent négligé : il permet pourtant de savoir ce qu'était censé faire le code à l'origine d'un bug par exemple

# Centralized Version Control System

- Les CVCSs proposent de résoudre le problème de collaboration (ou concurrence) par un point central faisant le versioning



- Connexion au serveur pour effectuer des échanges
- Une modification est prise en compte lorsque les données sont poussées vers le serveur
- Cependant, une coupure de réseau signifie qu'il n'est plus possible de travailler
- Un crash du serveur (ou perte de données) est synonyme de catastrophe
  - Sauf si le serveur est lui même versionné...

# 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)



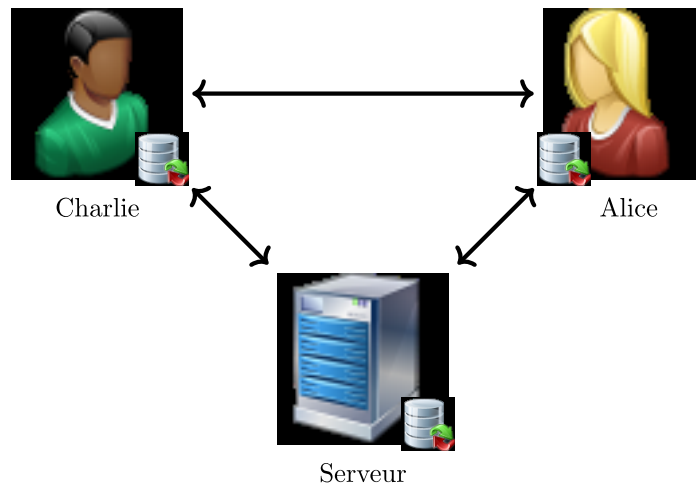
- Projet de l'ASF depuis 2009, créé en 2000 par l'entreprise CollabNet dans la vue de faire un successeur amélioré de CVS

# 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) 
    - Créé par Martin Pool en février 2005 pour Canonical. Version 0.0.1 sortie le 26 mars 2005
  - Hg (Mercurial) 
    - Créé par Matt Mackal le 19 avril 2005, en réponse à l'arrêt de l'offre gratuite de BitKeeper (un DVCS propriétaire)
  - git (Git) 
    - Créé par Linus Torvalds le 7 avril 2005, en réponse à l'arrêt de l'offre gratuite de BitKeeper (oui, le même)

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

# Avec qui échanger ?

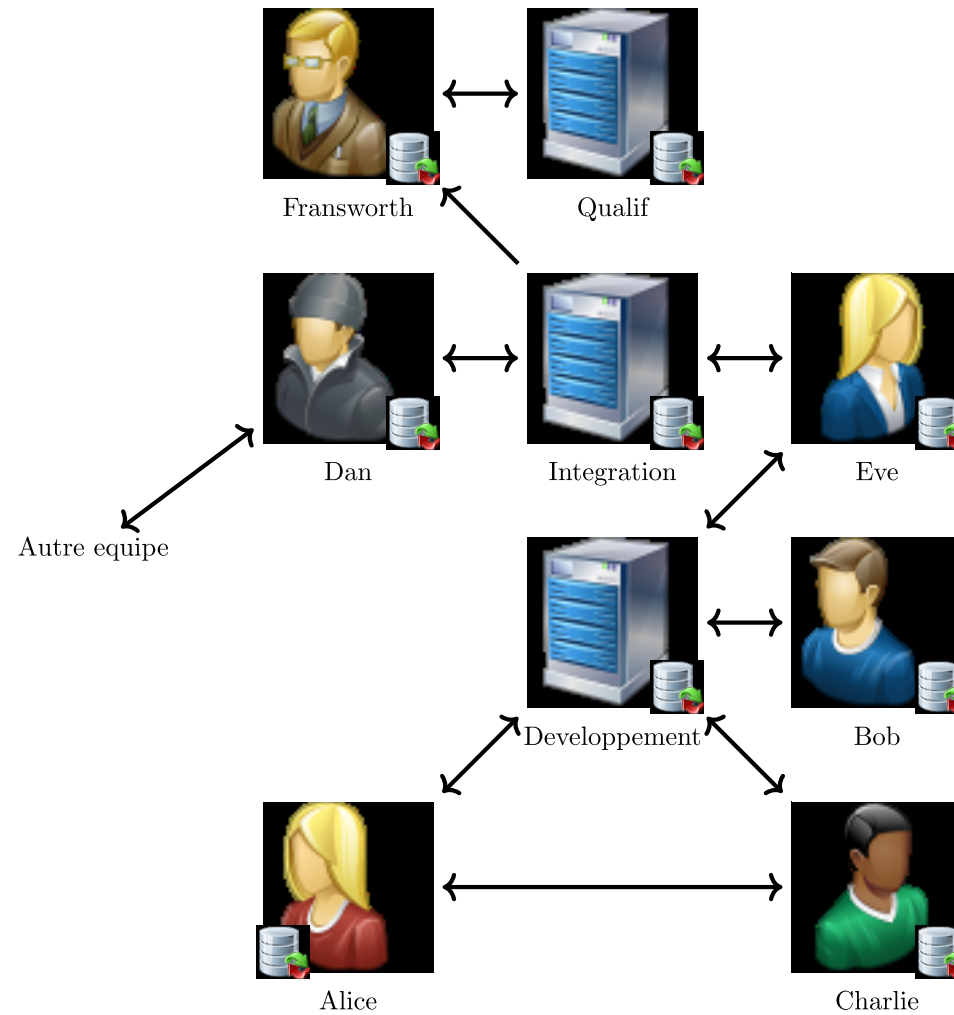
- Le système centralisé donne l'impression qu'il faut un seul dépôt
  - C'est faux, il suffit d'avoir des dépôts de **confiance**
  - Il n'est pas obligatoire de faire confiance à tout le monde
  - Il n'est pas non plus obligatoire de ne faire confiance qu'à un seul dépôt



- On observe en général dans les structures utilisant des DVCSs deux types de dépôts :
  - Personnel/local : dans lequel l'utilisateur travaille
  - Public/online : dans lequel un ou plusieurs utilisateurs publient du contenu
- Il est possible d'avoir plusieurs workflows différents selon les besoins




# Types de workflow possibles



# Distributed !

- Les dépôts sont complètement distribués
  - La notion de dépôt central disparaît donc, remplacé par un/des dépôts de **confiance** : professeur Fransworth ne fait confiance qu'au dépôt d'intégration
  - Chacun est **indépendant**, mais peut travailler avec les autres : Charlie et Alice travaillent ensemble, et Bob n'est pas impacté
  - Les dépôts sont toujours **stables** : le travail de l'équipe de Dan n'est redescendu sur le serveur de développement qu'après intégration
- Si ce workflow ne convient pas, beaucoup d'autres peuvent être créés selon les besoins
  - Ex : workflow du kernel Linux

# Comment échanger alors ?

- Une fois que Dan et Eve ont fait leur intégration, le but est de remonter l'information au professeur Fransworth
- Cette remontée d'information est une ***pull request***
  - Chacun est libre, on ne peut pas "forcer" quelqu'un à récupérer du contenu
  - À la place, on lui propose de récupérer les dernières modifications, ce qu'il fera, ou pas
- Cette demande peut être faite via : 
  - un mail/téléphone/vive voix (le plus humain/commun)
  - un échange d'un fichier .patch contenant les modifications
  - un système automatisé
  - ce que l'on veut

# Patchs "non-officiels"

- Charlie a fait une modification qu'il pense intéressante, mais il préfère que quelqu'un d'autre teste **avant** de rendre ça public
  - Alice peut directement récupérer ces modifications depuis le dépôt privé de Charlie
  - Alice peut appliquer ses propres patchs sur l'idée de Charlie pour réparer quelques soucis
  - Charlie récupère ensuite les patchs d'Alice pour publier le tout
- Faire des revues de contenu avant de publier est simple et permet d'éviter des erreurs
- Le dépôt n'est pas "pollué" par le travail en cours, tout est publié quand c'est prêt, et en attendant tout est **versionné**

# One more thing

- Les dépôts sont 100% indépendants
  - Ils doivent donc tous contenir **toutes** les informations
  - L'historique des commits, les logs, tout est disponible en local



- La **première** copie peut être longue (il faut tout récupérer)
- Le travail en local sera **instantané**, récupérer des logs, revenir sur des versions précédentes prend moins d'une seconde

# En bref

- Les DVCSs 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 !



# Fonctionnement de Git



# Plan

- Introduction
- *Fonctionnement de Git*
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- Configuration et outils externes

# Historique

- Le kernel Linux était versionné sur le DVCS propriétaire BitKeeper
  - Après des années de trolls/flames, BitKeeper et Linux ont préféré se séparer, l'annonce officielle date du 6 Avril 2005
  - 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 avril, le kernel linux est versionné sous Git



- Il aura fallu trois jours de développement avant d'avoir un système de versionning
- Linus Torvalds n'est pas un génie (enfin si), il a juste repris le problème de versionning de la base

# Le problème à résoudre

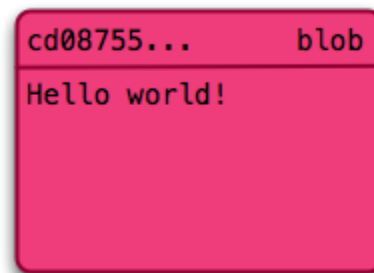
- Pour rappel : le système de versionning a pour but de sauvegarder et d'historiser du **contenu**
  - Pas des fichiers, ni une arborescence, mais bien du contenu
  - Chaque version différente = un contenu différent
- L'idée est maintenant de pouvoir identifier un contenu par rapport à un autre
  - La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

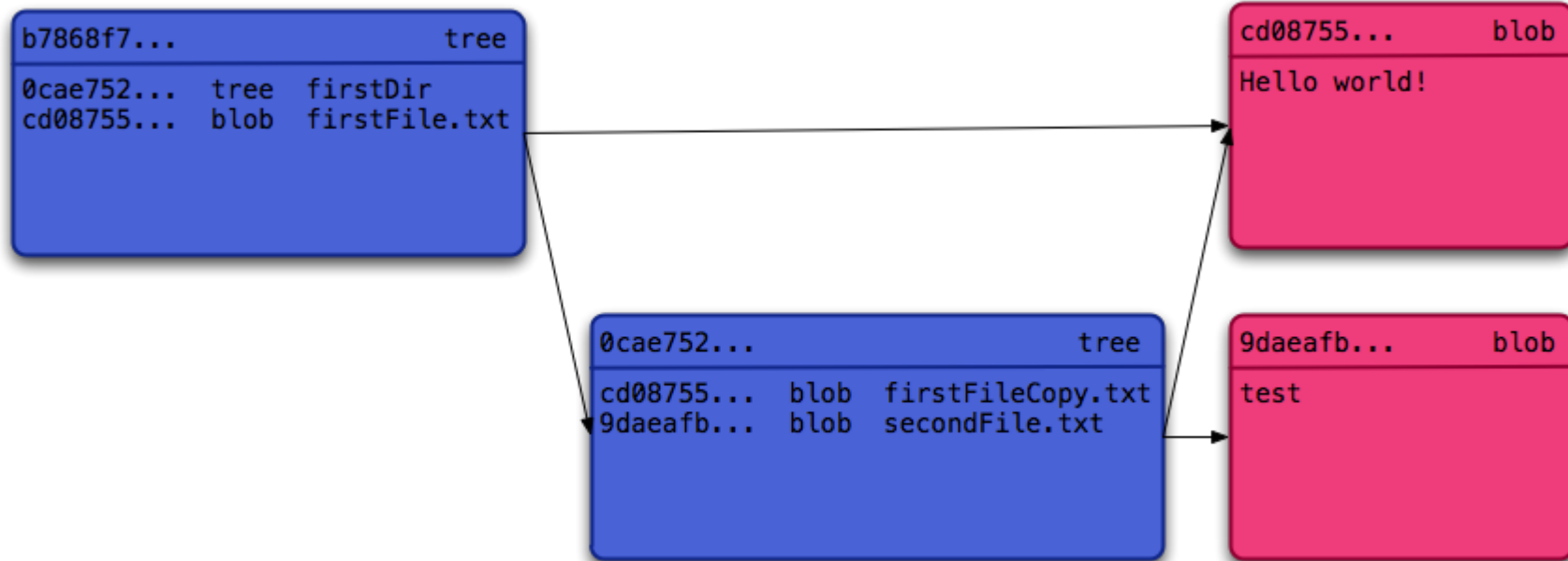
b7868f7...	tree	
0cae752...	tree	firstDir
cd08755...	blob	firstFile.txt

```
$ 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 seul 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

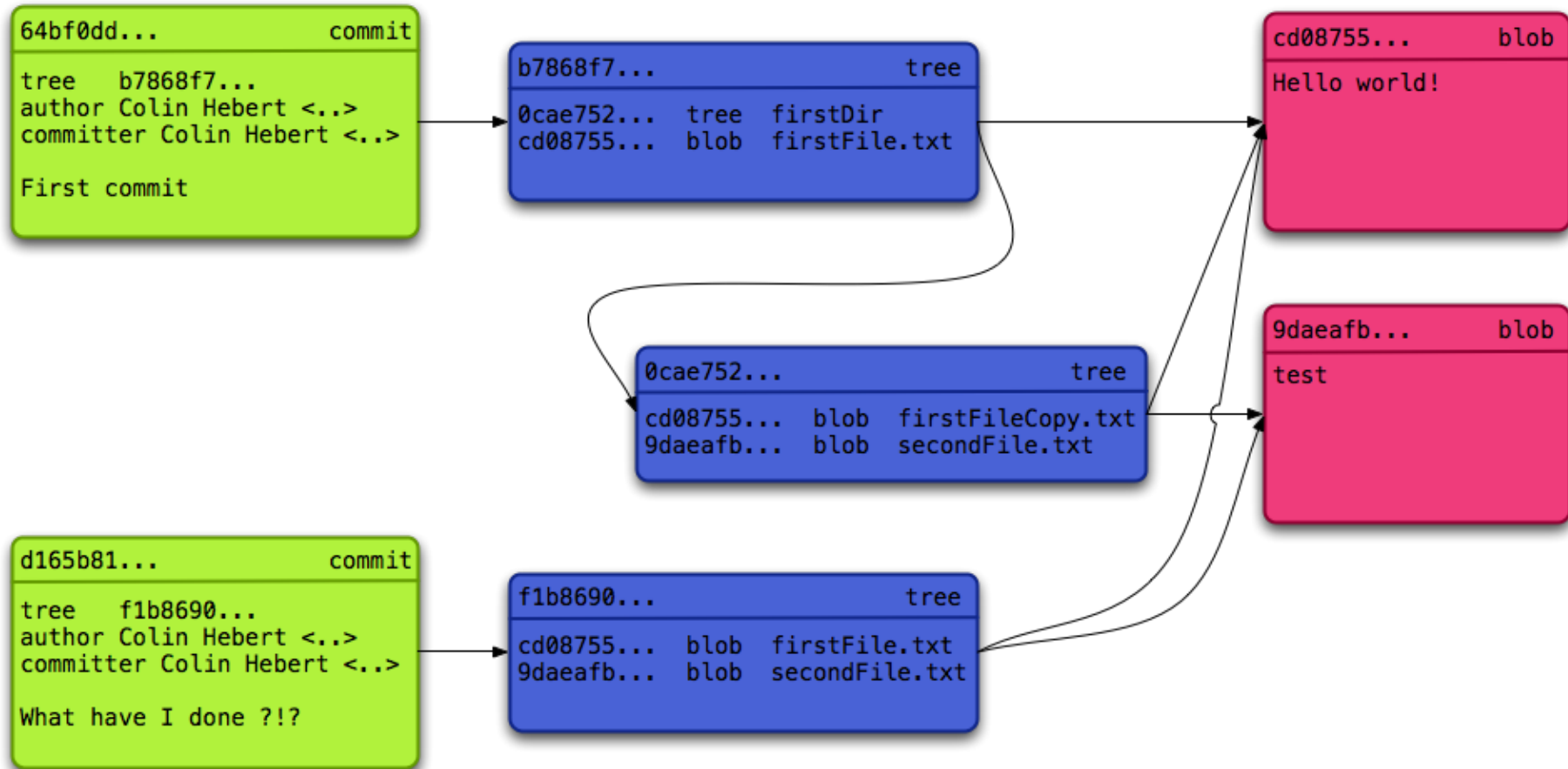
First commit
```



# 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

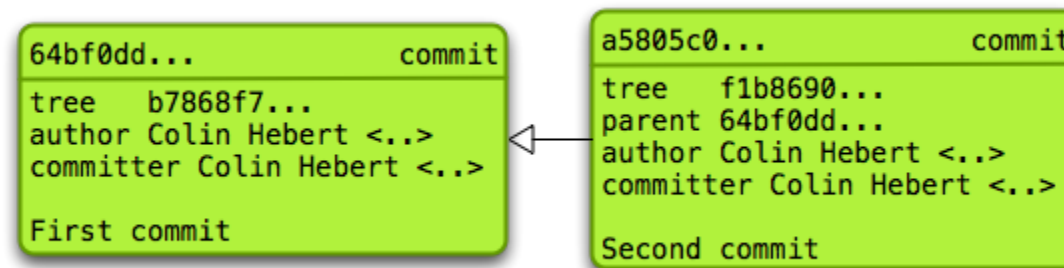
# Plein de commits



- 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)



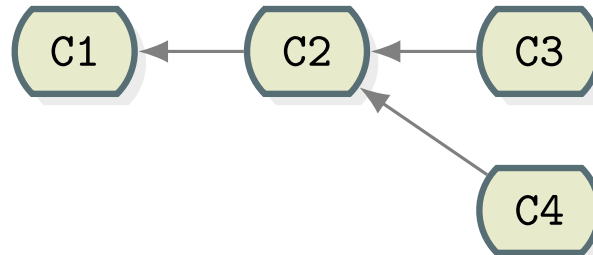
- Contrairement à d'autres VCS l'ordre de parution des commits n'influe pas sur l'ordre réel des commits
- Chaque commit sait d'où il "vient", mais ne sait pas vers où il "va"
  - Il pourrait y avoir plusieurs branches (ou aucune)
  - Quand le commit est fait, il n'y a pas de commit suivant !

# Les branches

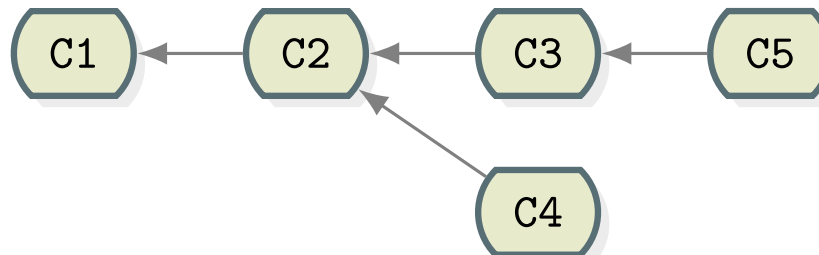
- Un concept important, parfois évité par les utilisateurs d'anciens VCS, est remis au goût du jour : les **branches**
- Certains VCS se basent sur le principe qu'une branche n'est qu'un dossier dans lequel on recopie tout l'ancien projet
  - L'historique n'est pas/plus accessible
  - Le changement de 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

# Les branches "proprement"

- À la base les branches sont simples, ce sont des commits pointant vers un même parent



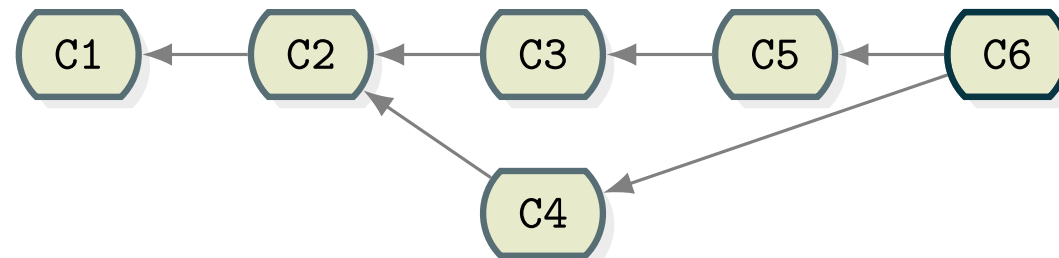
- Ce système permet d'avoir deux branches totalement indépendantes. **C3** et **C4** n'ont strictement aucun lien entre eux



- C5** n'est pas après **C4** même si chronologiquement les commits se suivent

# Les merges

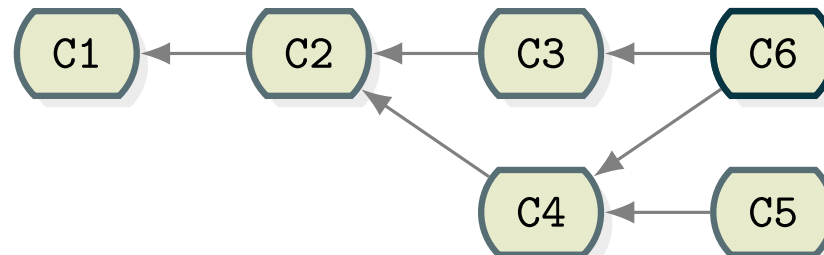
- Les branches ne sont que des commits distincts (non successifs) sur un graphe
- Merger signifie simplement créer un commit regroupant les deux (ou plus) branches



- 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

# Cas "amusant"

- Les commits successifs **C1**, **C2**, **C3** sont créés
- **C4** est une branche de **C2** et **C5** est le commit suivant **C4**
- Comment merger les deux branches sans merger le contenu de **C5** qui est un commit instable (ou non intéressant)
- Ce genre de cas est quasiment ingérable avec certains VCS, pourtant la solution est très simple



# 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 : vi, 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, le contenu de ce dossier sera analysé en détails

# 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](mailto: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





# 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

# 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

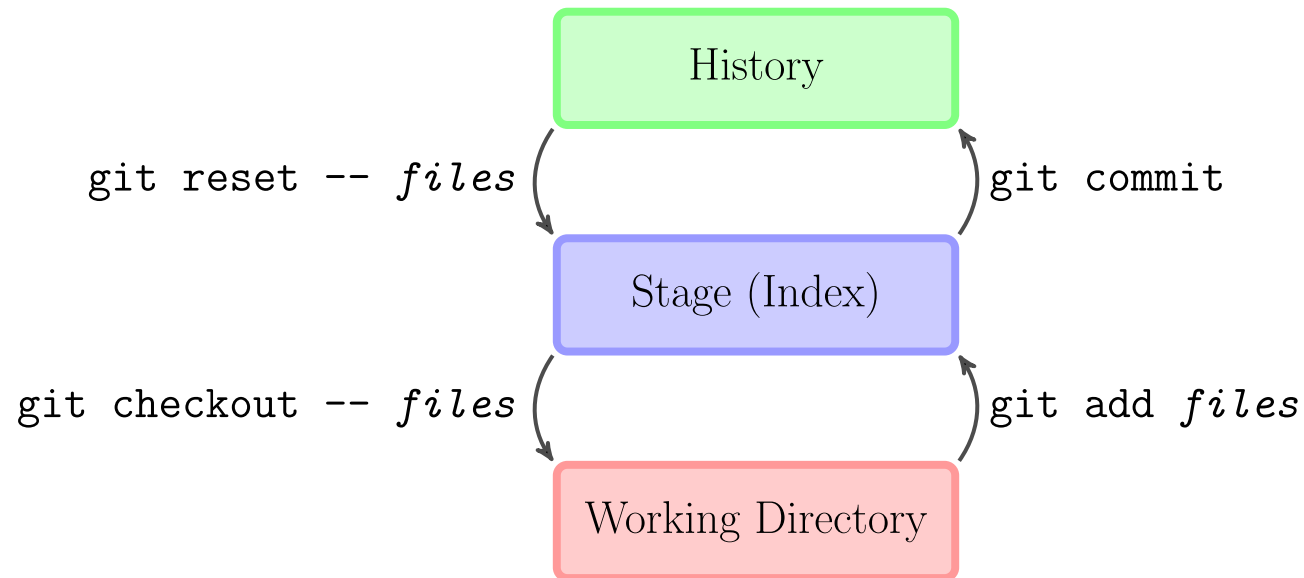


- Étapes de création du commit
  - Création/Modification/Suppression de contenu
  - Insertion des changements dans l'index
  - Validation de l'index dans l'état actuel et création de l'objet commit



# 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 committées

```
$ 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 checkout -- <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 reset 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 patches
  - 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 Fileglobs 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 reset 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 checkout -- <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 reset 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 Fileglobs 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!
+Goodby 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 essaiera 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

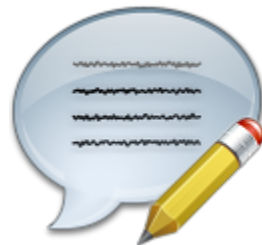
- L'appel de `git commit` avec l'option `-m` permet de tout faire en une seule ligne de commande

```
$ 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 éliminant les données inutiles

```
$ git log --oneline
7cc6ab3 A relevant messagea
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

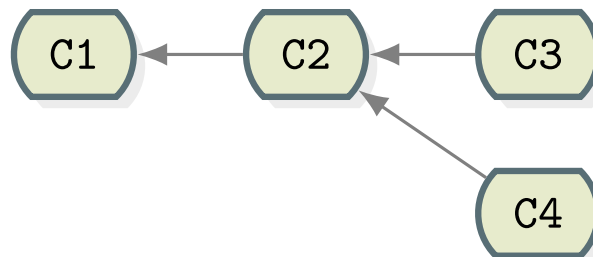
- La commande **git checkout** permet de revenir à l'état de n'importe quel commit effectué

```
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 -b with the checkout command again. Example:
    git checkout -b new_branch_name
HEAD is now at a5805c0... Second Commit
$ ls
FirstFile.txt secondFile.txt
```





## Lab 2

# Les références

# Plan

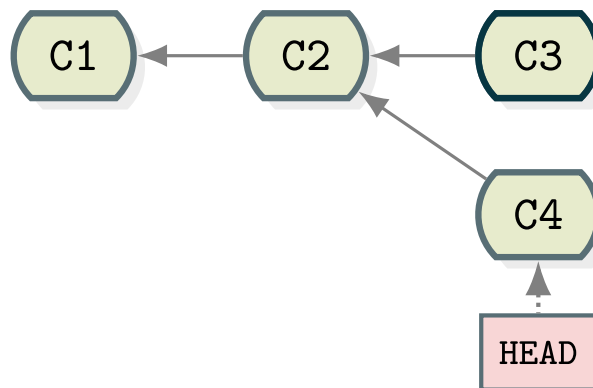
- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- *Les références*
- Utiliser Git en distant
- Configuration et outils externes

# 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**, référencent 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

# 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

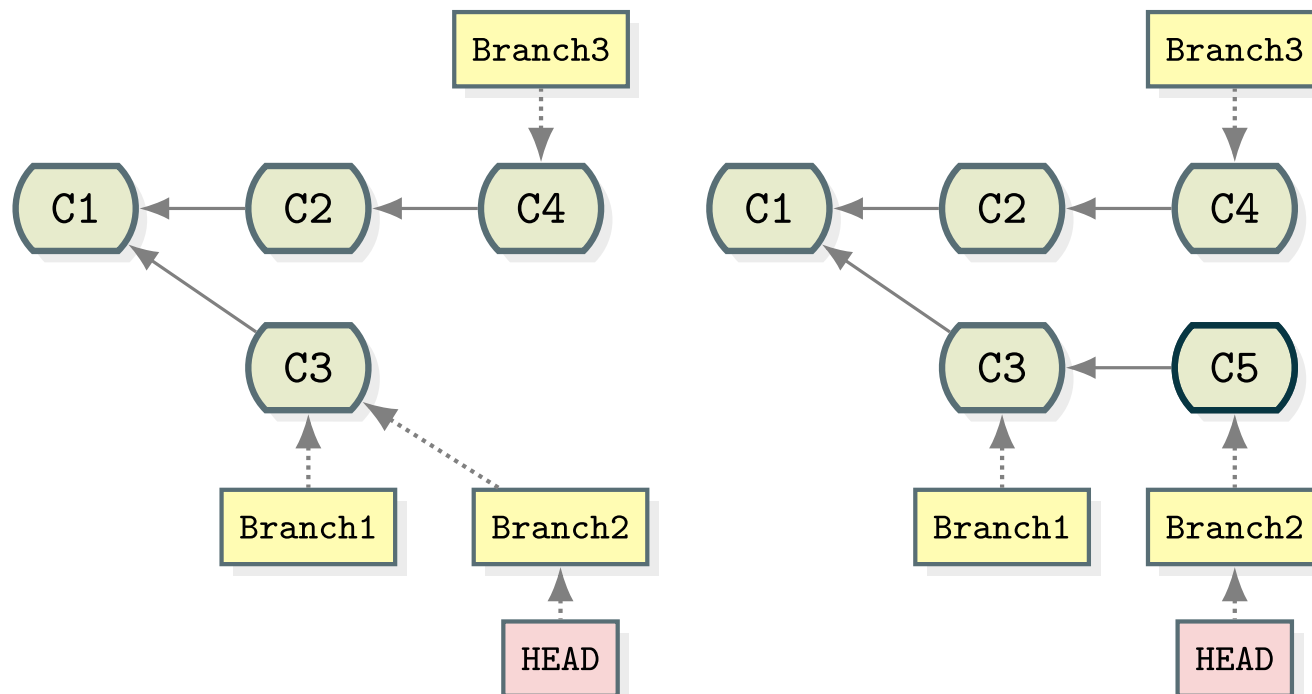


- La référence **HEAD** représente toujours le commit sur lequel les modifications sont basées
  - **HEAD** peut référencer un commit directement (detached head) ou une autre référence
  - En cas de **git checkout**, **HEAD** référence l'élément spécifié



# Les branches

- Les branches sont des références *évoluant* lors du commit
- Si **HEAD** référence **B2** (qui référence **C3**), après le commit de **C5**, **B2** référencera **C5**. **HEAD** continuera de pointer vers **B2**



- 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

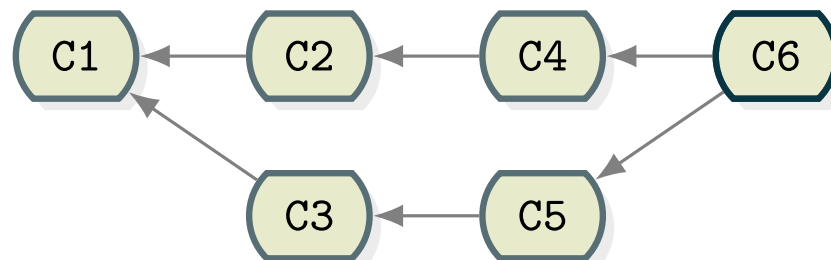
```
$ git log --graph --all --oneline --decorate
* b3d9655 (HEAD, master) Merge branch 'secondBranch'
| \
| * dc1ce86 (secondBranch) Foo bar modification
* | 7cc6ab3 A relevant message
* | a5805c0 Second commit
| /
* 64bf0dd First commit
```



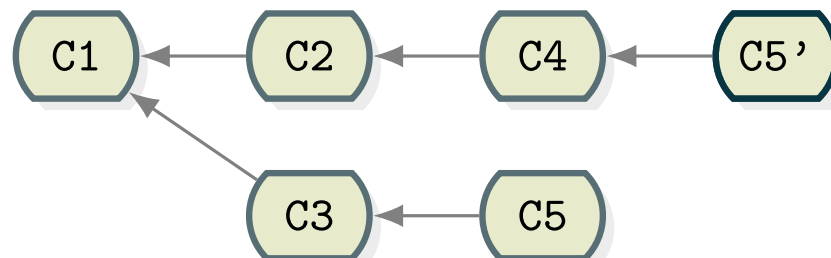
**TP3.1 ⇒ TP3.3**

# Fusion et déplacement de commits

- 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

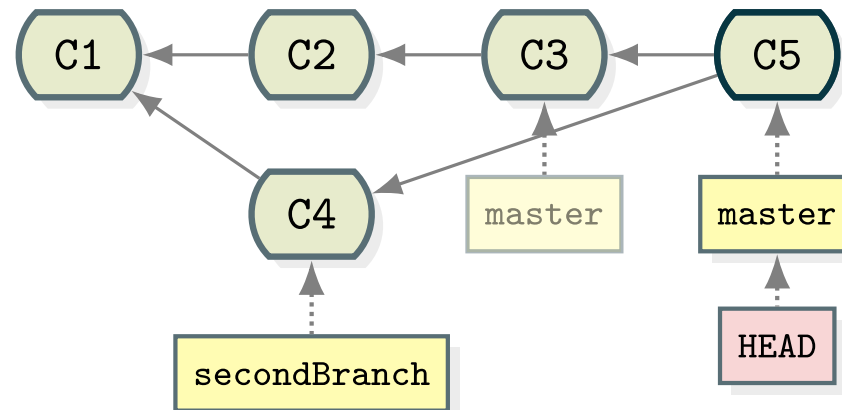


- Il est possible de rejouer des commits à partir d'un autre commit
  - `git cherry-pick` récupère les modifications apportées par un ou plusieurs commits et les ré-applique sur **HEAD**
  - `git rebase` "déplace" et modifie une suite de commits



# git merge

- L'opération **git merge** prend en paramètre un (ou plusieurs) commit à fusionner avec le **HEAD** actuel



# 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")
```

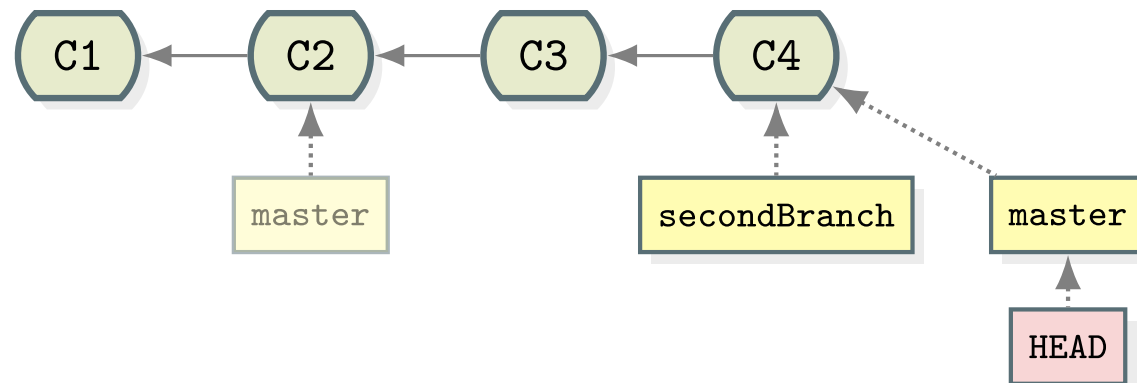
```
$ cat firstFile.txt
<<<<<<< HEAD
Hello world!
Test
=====
Foo bar
>>>>>>> secondBranch
```

# Résolution de conflits



# 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



# Copie de commit

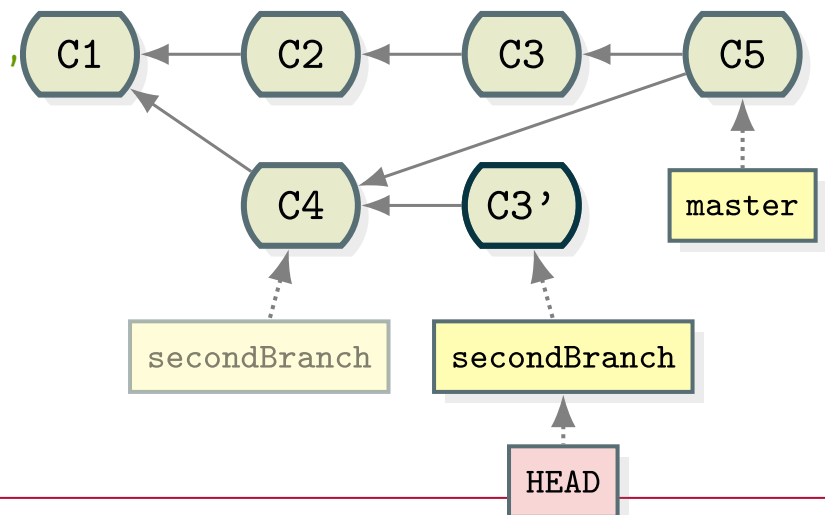
- Il est possible de copier un commit vers un nouvel emplacement
  - L'opération consiste à récupérer les modifications apportées par un ou plusieurs commits et les rejouer sur le nouvel emplacement, avec les mêmes messages
  - Les anciens commits sont toujours présents
  - Les nouveaux commits (les 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

```
$ 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 ⇒ TP3.5**

# rebase

- 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 le quel 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>

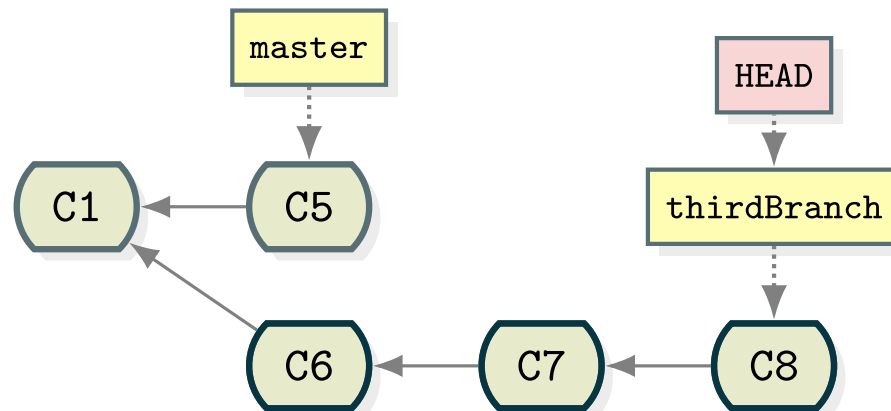
Commit/référence déplacée

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
```

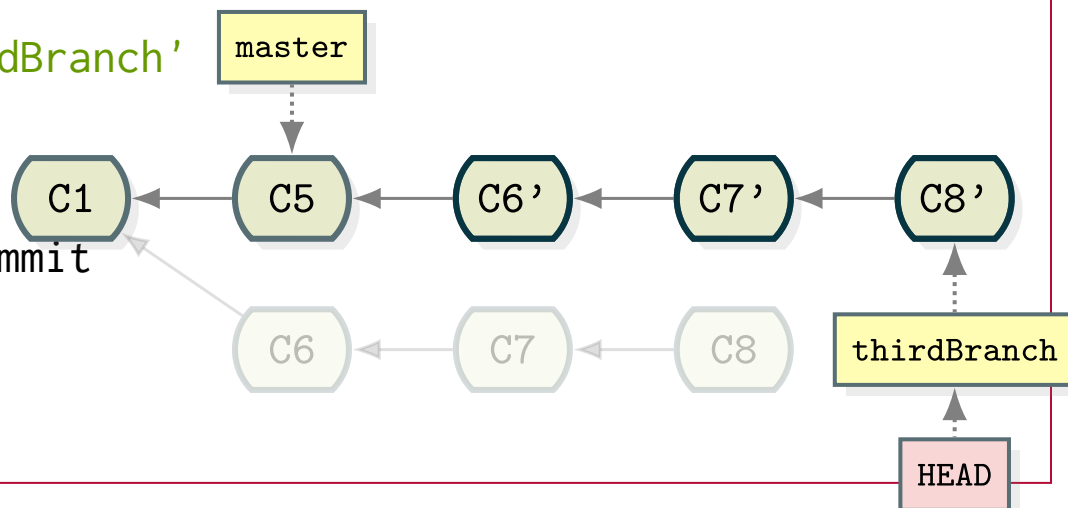
# git rebase

- 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



# 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
```



# TP3.6

# 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
    - Créateur 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
- Seul le tag annoté est enregistré ainsi. Un tag standard est juste compté comme une référence

```
8fb5a89...      tag
object b3d9655...
type commit
tag firstTag
tagger Colin Hebert <..>

An annotated tag
```

```
$ 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
```

- 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

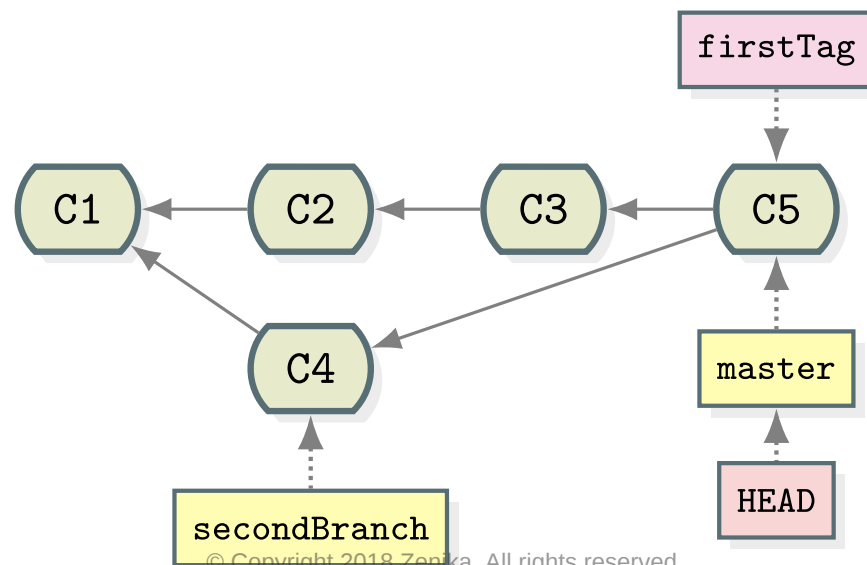
- m <msg>  
Spécifie le message lors de la création d'un tag annoté
- <tagname>  
Nom du tag à créer/supprimer/vérifier
- <object>  
Élément à tagger. Si non spécifié le commit de HEAD est utilisé

- 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**



- Le système est simple, les contenus des éléments indexés et non-indexés sont commités au dessus de **HEAD** et référencés dans le **stash**
  - Seul le **stash** référence ces commits, s'ils sont supprimés du **stash**, ils sont dangling et supprimés à terme de la base
  - Ce sont bien 2 commits qui sont créés
    - 1 pour les éléments indexés
    - 1 pour les non indexés



# 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

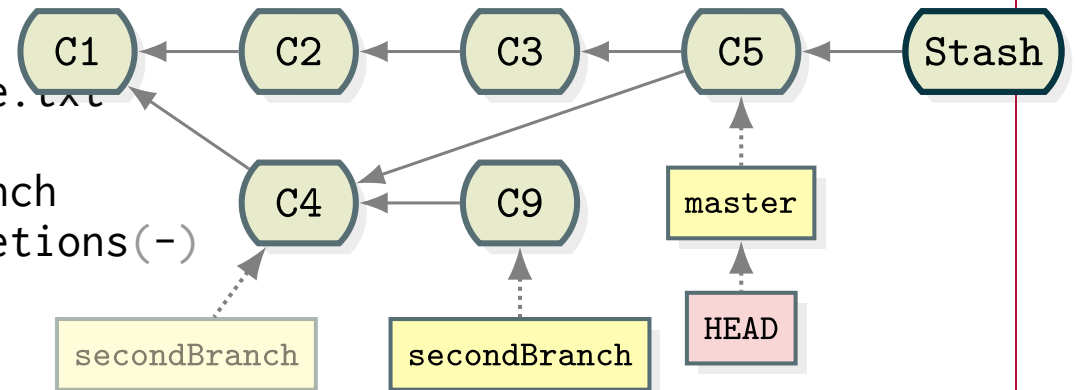
<stash>

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 ne sont partagés et sont strictement personnels
- 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

# Références avancées

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

# 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  $\Delta$ )
  - `<ref>^@` Tous les ancêtres de `ref` (mais pas `ref`)
  - `<ref>^! ref` mais pas ses ancêtres



## Lab 4

# Utiliser Git en distant

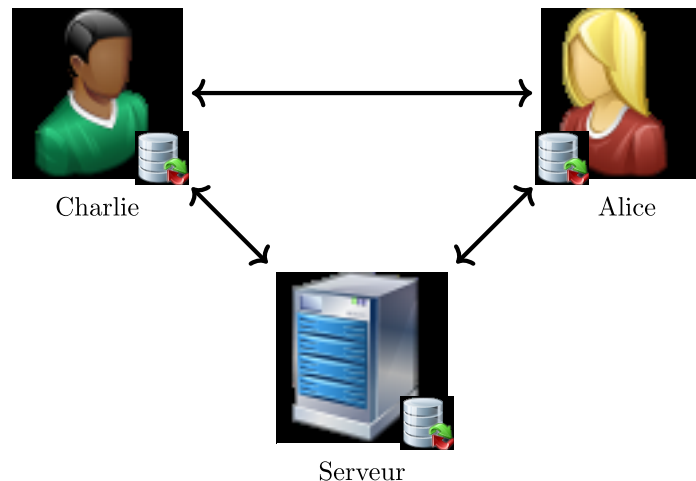
# Plan

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- *Utiliser Git en distant*
- Configuration et outils externes



# 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 des dépôts pour échanger les différentes modifications qui ont eu lieu



- Les dépôts uniquement distants, peuvent être créés en **bare**
  - Accessibles depuis l'extérieur
  - Contiennent uniquement la base de données de Git

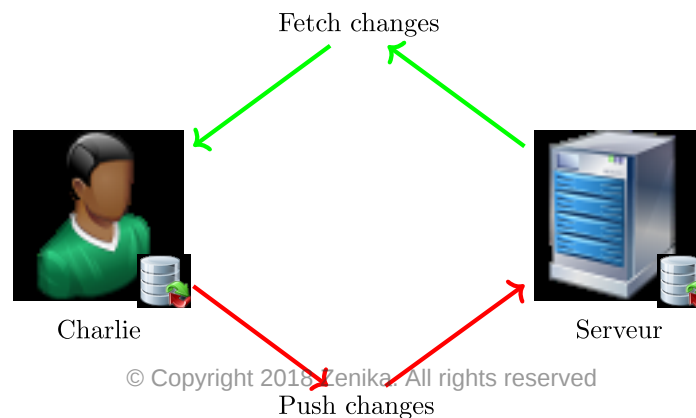
# Types d'accès

- 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



# Déclaration d'un dépôt externe

- La déclaration et gestion des dépôts externes se font 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 patches
- `git diff` permet de générer des patches

```
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**





## Lab 5

# Configuration et outils externes

# Plan

- Introduction
- Fonctionnement de Git
- Utiliser Git en local
- Les références
- Utiliser Git en distant
- *Configuration et outils externes*

# 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

# Les alias

- 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
```



# 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 sur `.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 #{$ancienrev}..#{$nouvellev}`.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

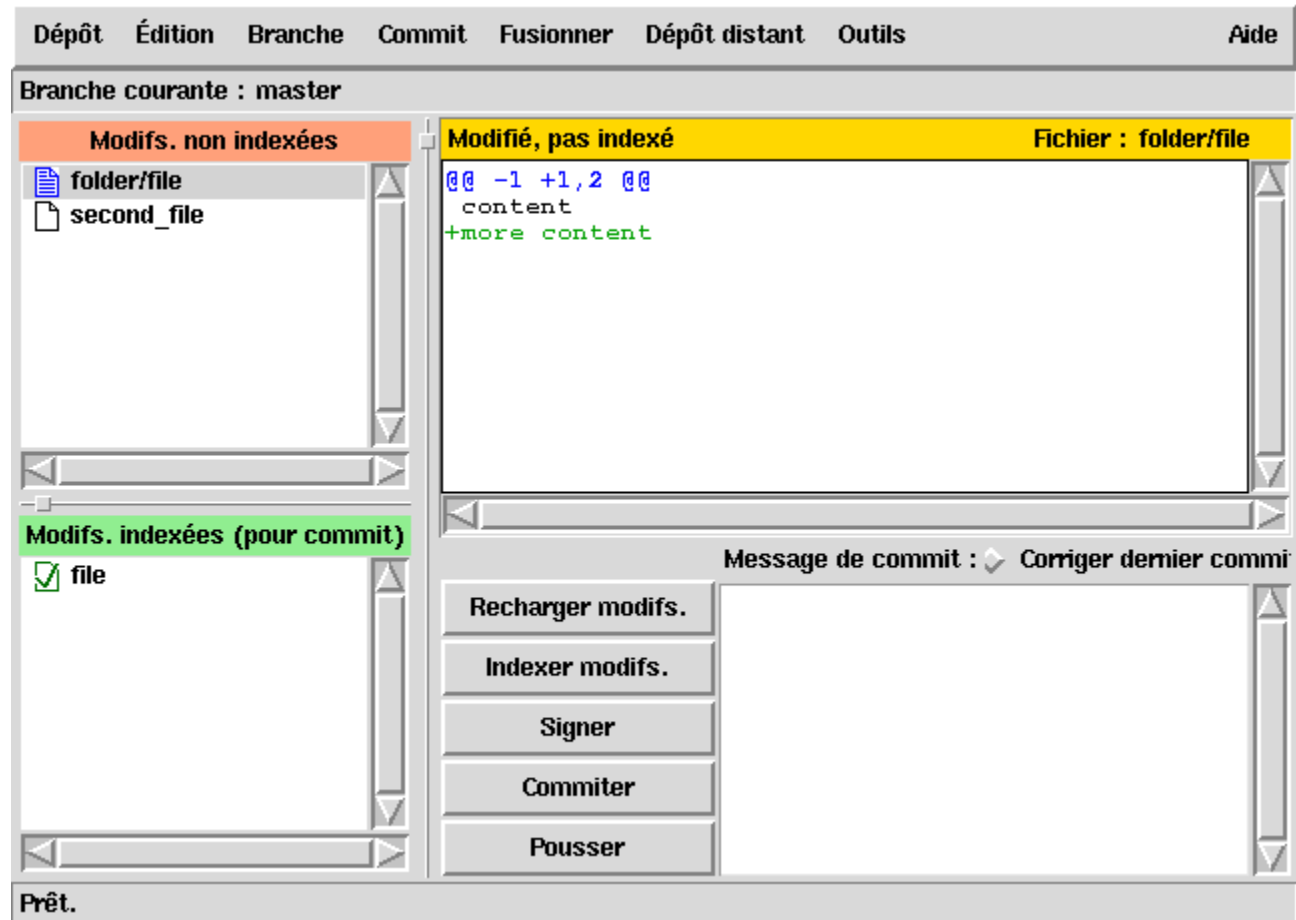
- Git est utilisable comme client valide d'un serveur SVN avec les commandes de type `git svn`
- Avantages
  - Permet de profiter localement des avantages de Git
  - Passage progressif au fonctionnement de Git
- Désavantages
  - Version tronquée de Git
  - Préférable de garder une structure linéaire

# Complétion/couleurs/éditeur

- Un fichier de complétion bash `git-completion.bash` est fourni dans l'installation de Git et permet compléter les commandes grâce à la touche `tab`, ZSH supporte la complétion 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`

# GUI

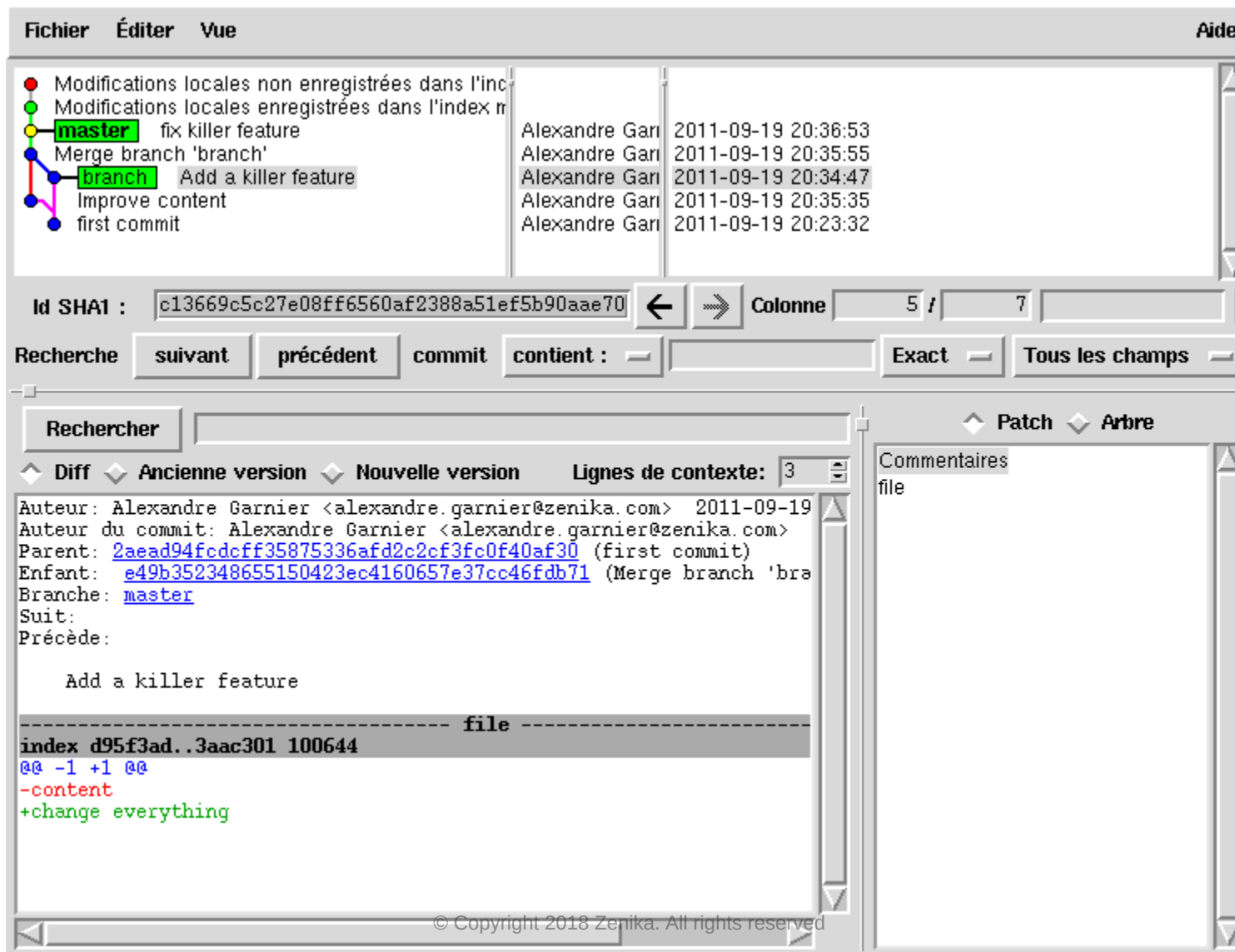
- Git fournit une interface graphique par défaut : `git gui`



# GUI

- `git-gui` permet en direct :
  - De voir l'état de l'arbre de travail
  - De voir l'état de l'index
  - De voir les différences de chaque fichier
  - De commiter
- Mais il permet aussi :
  - De gérer les branches
  - De gérer les dépôts distants
  - D'effectuer des merges

- Git fournit un autre outil graphique pour le parcours de l'historique : **gitk**





# gitweb

- De même, il existe une interface web par défaut à Git : **gitweb**

Fichier Édition Affichage Historique Marque-pages Outils Aide

[projects](#) / [.git](#) / **summary** git

summary | [shortlog](#) | [log](#) | [commit](#) | [commitdiff](#) | [tree](#)  <sup>?</sup> search:  ☐ re

description Unnamed repository; edit this file 'description' to name the repository.  
owner Alexandre Garnier  
last change Mon, 19 Sep 2011 18:36:53 +0000

**shortlog**

35 min ago	Alexandre Garnier	<b>fix killer feature</b>	master	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
36 min ago	Alexandre Garnier	<b>Merge branch 'branch'</b>		<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
37 min ago	Alexandre Garnier	<b>Improve content</b>		<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
37 min ago	Alexandre Garnier	<b>Add a killer feature</b>	branch	<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>
49 min ago	Alexandre Garnier	<b>first commit</b>		<a href="#">commit</a>   <a href="#">commitdiff</a>   <a href="#">tree</a>   <a href="#">snapshot</a>

**heads**

35 min ago	<b>master</b>	<a href="#">shortlog</a>   <a href="#">log</a>   <a href="#">tree</a>
37 min ago	<b>branch</b>	<a href="#">shortlog</a>   <a href="#">log</a>   <a href="#">tree</a>

Unnamed repository; edit this file 'description' to name the repository.

# gitweb

- Lancement rapide via `git instaweb`
- Par défaut : `http://localhost:1234/`
- S'intègre parfaitement avec les serveurs HTTP
- Permet de :
  - Parcourir plusieurs dépôts
  - Consulter l'historique
  - Accéder aux trees et blobs
  - Voir les diffs

# Gerrit

- Gerrit est une interface web de relecture de code
- Il a été développé par Google dans le cadre du développement d'Android
- Le principe est le suivant :
  - Un développeur pousse ses modifications sur le dépôt Gerrit
  - D'autres développeurs peuvent visionner et commenter les modifications et les approuver ou non
  - Le développeur peut repousser des corrections si nécessaire
  - Une fois les modifications approuvées, celles-ci peuvent être intégrées dans le dépôt de confiance





# Lab 6