Le gestionnaire de versions Git

Gilles Maire

2017

Plan de la formation

Présentation

Rubriques

- Définitions
- Comparaison des gestionnaires de version
- Interface graphique
- Première création d'un dépôt par clonage

Définitions

Les gestionnaires de versions

- fournissent un dépôt conservant les fichiers dans toutes leurs différentes versions
- sont capables de gérer la communication entre un ou plusieurs postes de développement
- sont capables de récupérer la dernière version du dépôt ou une version donnée ce qui est très utile pour la maintenance des versions clients
- permettent à chaque développeur de mettre à jour une nouvelle version avec ses propres modifications et aux autres développeurs de retrouver ces modifications
- gèrent les conflits entre deux versions contradictoires fournies par deux développeurs
- ce sont donc des outils de travail collaboratifs, véritables mémoires des projets et aussi de bons moyens de backup.

Quelques remarques

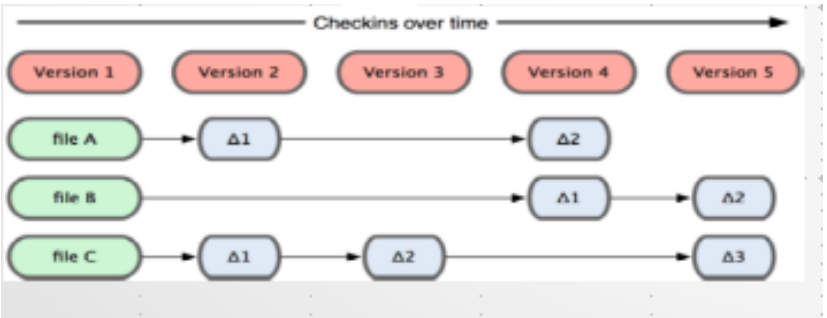
- Les gestionnaires savent stocker des fichiers sources ainsi que des fichiers binaires
- Par contre on évite de stocker les fichier exécutables, les objets ou les librairies provenant des sources (pour ne pas provoquer des mises à jour pléthoriques et inutiles)
- Pour les fichiers binaires on aura une politique de mise à jour tout ou rien
- Pour les fichiers textes, la politique de mise à jour est incrémentale via des commandes d'ajout, de suppression ou de modification de lignes

Dépôts publics/ dépôts privés

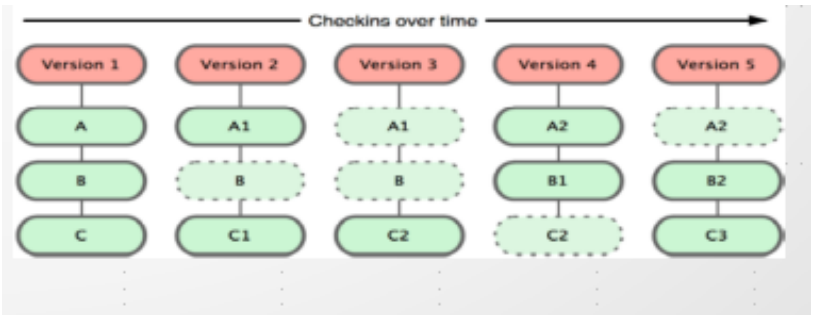
- Un dépôt peut être protégé par mot de passe ou par clé publique/privée, mais il peut être également en accès libre en lecture
- C'est le cas pour tous les projets open source dont les sources sont accessibles à tous.
- Un projet Open Source ne veut pas dire que vous pouvez modifier les sources sans vous être authentifié auprès de l'équipe de développement qui vous aura remis un mot de passe ou mieux qui aura accepté votre clé publique.
- Avec les principaux serveurs de dépôts de projets open source (GitHub, SourceForge, Bettercodes, GitLab, Gitorius) on peut accéder à des sources sans mot de passe, ils hébergent des projets publics
- D'autres serveurs de dépôts ou même certains des dépôts précédents permettent de développer des projets privés sous condition (Bittbucket, GitEntreprise).
- voir la liste sur https://git.wiki.kernel.org/index.php/GitHosting

Histoire de Git

- Git a été écrit en 2005 par Linus Torvalds en remplacement de Beetkeeper qui tentait une percée commerciale.
- Git exécute la plupart de ses commandes localement sans avoir besoin de communiquer avec un serveur
- Pour chaque version, git maintient les fichiers soit sous forme de pointeur si les fichier n'ont pas changé, soit sous forme d'instantané si le fichier a changé (c'est à dire que tout le fichier est copié et non pas les delta)



Anciens systèmes



L'originalité de Git

- Git a été conçu comme un système de fichiers
- Git est décentralisé
- Git peut utiliser un port Git (9418) mais aussi un port ssh (22) ou http (80)
- Git possède deux structures de données :
  - le cache de répertoires
  - une base d'objets
- La base d'objets comprend 4 types d'objets :
  - **blob :** représente le contenu d'un fichier
  - **tree :** liste d'objets de type blobs et informations associées à chaque blob
  - **commit :** donne l'accès à l'historique de l'arborescence
  - **tag :** permet de donner un nom ou de marquer certains commit
  - **refs :** pointeur sur les objets

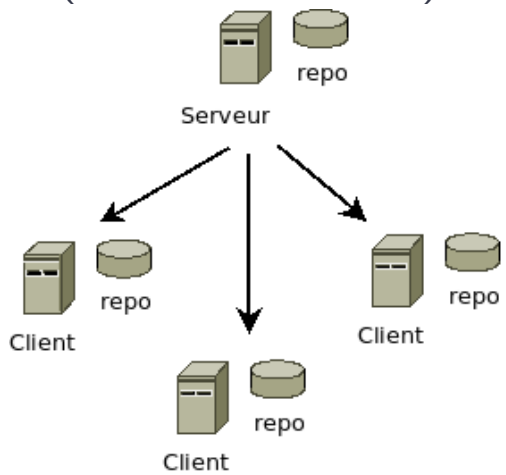
Avantage GIT

- Si un serveur est défaillant chaque serveur et chaque client dispose de l'ensemble des versions et pas uniquement de la dernière version
- L'architecture est prévue pour que les mises à jours du logiciel Git lui même se passent bien.
- Git ne fonctionne pas en différenciel mais en instantané
- Git vous signale les fichiers de l'arborescence non pris en compte, c'est utile contre les oublis de fichiers
- Git est très rapide, il est de conception simple, il permet des milliers de branches parallèles
- Git compacte les données de façon très efficace.
- La gestion des branches est très légère.
- Un serveur centralisé Git peut être utilisé comme serveur de référence, il sert en même temps de serveur de backup (il ne contient rien de plus que chacune des versions locales)

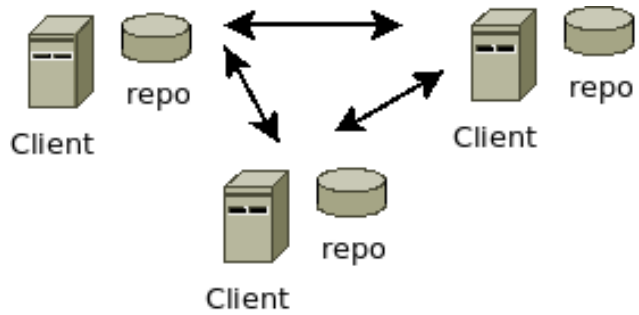
Comparaison des gestionnaires de version

Gestionnaires centralisés vs décentralisés

Centralisés (SVN, CVS, SCCS)



Décentralisés (Git, Mercurial)



Les différents gestionnaires

- **RCS :** (Local) base de données de modifications
- **CVS :** (Centralisé) plus très utilisé, il possédait de nombreuses limitations comme l'impossibilité de suivre un fichier renommé
- **SVN :** (Centralisé) encore utilisé il est centralisé il corrige les défauts de CVS
- **Mercurial :** (Distribué) développement commencé en même temps que git, il est distribué et non plus centralisé. Mozilla, Python, OpenOffice l'utilisent
- **GIT :** (Distribué) nouveau et de plus en plus utilisé, distribué, il est utilisé pour maintenir les sources du noyau Linux, Debian, VLC, Android, Gnome, Qt
- **Bazaar :** (Distribué) sponsorisé par Ubuntu il est décentralisé. Il est utilisé par Inkscape, mySQL, Ubuntu

Utilisation la plus basique

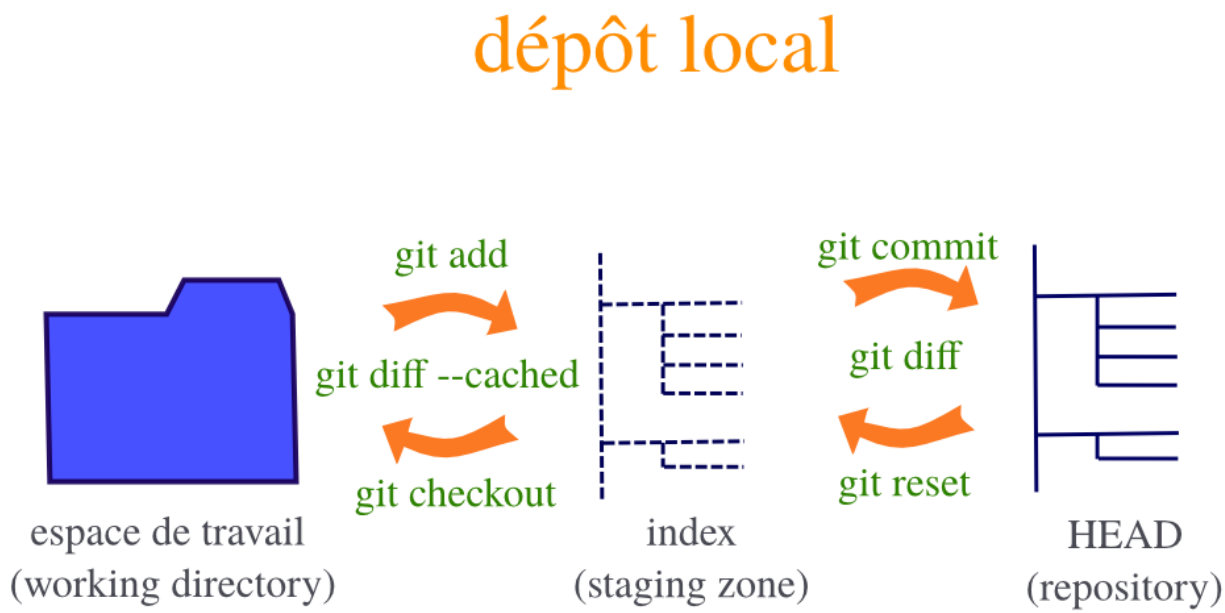
- bazaar
  - bzr checkout adresse
  - mise à jour : bzr pull
- mercurial
  - hg clone adresse
  - hg pull
- GIT
  - git clone adresse
  - mise à jour : git pull
- CVS et SVN:
  - cvs checkout adresse
  - mise à jour : cvs update

Git caractéristiques

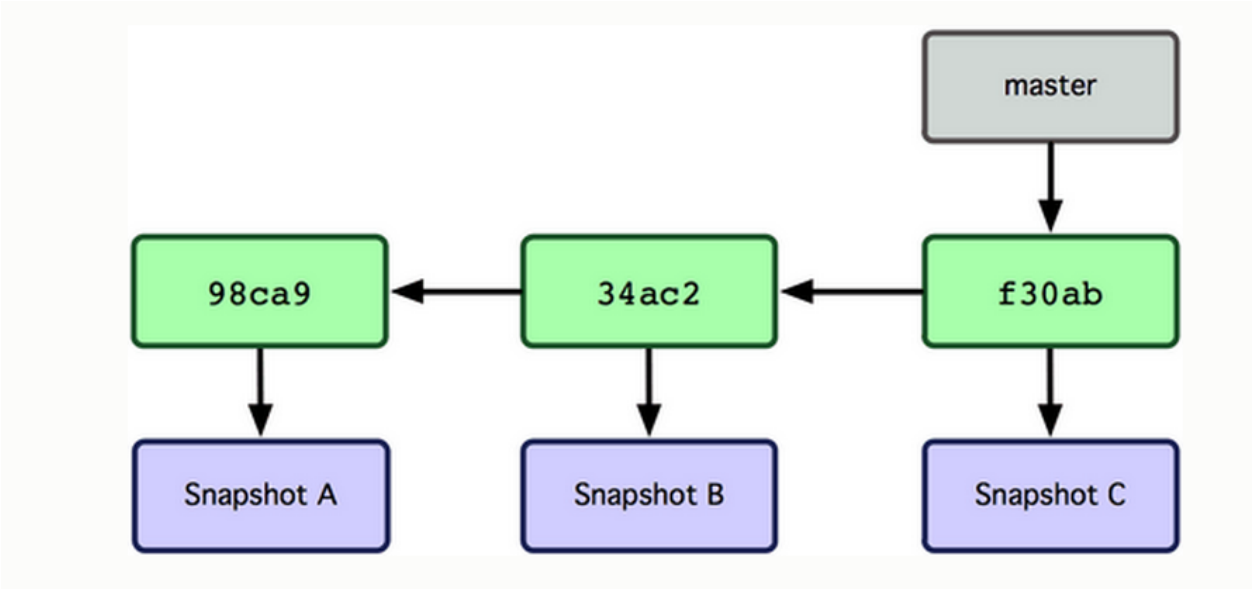
- Git stocke en local l'historique du projet
- Git peut faire des opérations de reversing sans interroger de serveur
- En avion ou en train vous pouvez soumettre vos modifications au dépôt local de votre ordinateur et synchroniser ces modifications avec d'autres serveurs plus tard
- Git utilise des sommes de contrôle SHA sur chaque fichier qu'il traite en communication avec un dépôt
- Chaque fichier peut avoir quatre états : validé (stocké en base localement), modifié (modifié mais non stocké en base), indexé (le fichier fera partie du prochain instantané) et non pris en compte par le système de version
- Un dépôt git peut en théorie être autonome mais on l'associe toujours à un serveur distant pour :
  - permettre la sauvegarde sur une autre machine
  - permettre à d'autres développeurs de se connecter dessus quand sa propre machine n'est pas connectée

Les arbres

- Un dépôt local est composé de trois «arbres» gérés par git :
  1. le premier est votre espace de travail qui représente vos fichiers
  2. le second est un Index qui joue un rôle d'espace de transit pour vos fichiers
  3. le commit où HEAD pointe vers la dernière validation que vous avez faite.



Les instantanés de Git



Interface graphique

Git graphique vs git en ligne de commande

- Git ayant été développé par le créateur de Linux, il est naturel que le mode ligne de commandes soit le mode initial de Git
- Un certain nombre d'outils graphiques sont disponibles qui permettent des manipulations globalement comparables à celles proposées par l'interface en lignes de commandes.
- Par contre n'oublions pas que Git peut évoluer, que vous pouvez tomber sur un projet sur un dépôt de git récent utilisant des changements de syntaxe. Dans le mode ligne de commande, la mise à jour se fera de façon transparente, avec une version graphique les choses peuvent être plus compliquées.
- Quand un message d'erreur apparaît avec une suggestion de réparation, des suggestions sont données par des commandes en mode ligne.
- Quand on ne comprend pas un message d'erreur bloquant, on trouvera la réponse dans des forums ou sur Google mais avec une explication en lignes de commandes
- Les modes graphiques proposent toujours/souvent un mode console pour interagir en mode ligne de commandes.
- Il est donc bon de connaître les grandes fonctionnalités en ligne de commande pour pouvoir s'y retrouver.

Peu importe l'interface

- Quel que soit votre choix d'interface, vous pourrez en changer à tout moment puisque chaque interface s'appuie sur le mode ligne.
- De plus en plus de projets regroupent des développeurs préférant MaxOsX, d'autres adoptant Linux et d'autres Windows. Chacun utilise l'interface qu'il préfère en mode ligne ou en mode graphique et travaillera sur les fichiers dans son environnement
- Il faut prendre conscience que les projets sous Windows auront des fins de ligne marquées par CR/LF et les Unixiens (Linux, MacOsX) auront des fins de ligne marqués par un CR. Git gère cela aussi.
- Les habitués aux interfaces svn sous Windows regretteront de ne pouvoir disposer d'un item aparaissant sur le clic droit des dossiers du gestionnaire de fichier pour lancer des opération git. C'est dû au fait qu'un répertoire git ne comprend des informations Git qu'à sa racine

Beaucoup d'IDE proposent un git intégré

- Qt (bibliothèque C++ fonctionnant sous Android, IOS, LinuxEmbarqué, Windows, Linux, MacOSX) dispose de Git dans son IDE QtCreator
- Eclipse propose Git en module souvent installé de base avec Eclipse

Via les commandes

- Obtenir l'aide sur la commande command

```
git help command
git help commit
```

- Depuis IRC-freenode :
- canaux
- #git
- #github

Première création d'un dépôt par clonage

Récupération via clone

- Pour récupérer les sources depuis un serveur Git distant au choix

```
# Pour créer un répertoire repertoire
git clone URLDepot repertoire
# créera le répertoire depot
git clone URLDepot
```

- Exemples :

```
 # création du répertoire monprojet
git clone git://github.com/schacon/grit.git monprojet
 # création du répertoire monrepertoire
git clone git://github.com/schacon/grit.git monrepertoire
 # création du répertoire depot (fin de l'arborescence)
git clone ssh://gilles@ignu.fr:/home/gilles/depot
 # création du répertoire mondepot
git clone ssh://gilles@ignu.fr:/home/gilles/depot mondepot
 # clonage d'un dépot local
git clone repertoire repertoire2
 # clonage via http
git clone http://gilles@ignu.fr:depot mondepot
```

Installation d'un client Git et clone d'un dépôt

- En fonction de notre environnement installer un client git
- Retrouver l'équivalent de la commande clone si vous utilisez une interface graphique

- Cloner le projet JQueryFastSite que vous trouverez sous sourceforge.net
- Examiner les fichiers du dossier
- Supprimer le dossier créé et refaites un clone via l'interface mode ligne
- Cloner le dépôt local que vous venez de créer dans un autre dépôt local.

Cas d'un clonage rapide

- On peut cloner un dépôt en spécifiant le paramètre –depth au nombre de versions que l'on souhaite télécharger
- Ceci est utile :
  - pour limiter le temps de chargement à la version courante et pas à l'ensemble des révisions
  - pour visiter la version actuelle d'un logiciel
- Ceci ne permet pas d'utiliser toutes les fonctionnalités de Git
- La syntaxe de la commande git avec profondeur de 1 est :

```
git clone --depth=1  addressedepot
```

Base

Rubriques

- Exploration
- Status et log
- Démarrer un dépôt vide

Exploration

Le répertoire .git

- En explorant le répertoire de votre projet, vous constaterez un répertoire nommé .git
- Si vous ne le voyez pas vous devez activer l'affichage des fichiers cachés.
- D'autres fichiers cachés peuvent apparaître à l'extérieur de ce répertoire .git, ils seront explicités plus loin
- Si vous supprimez le répertoire .git, vous vous retrouvez avec les sources de votre dépôt, sans que ces sources ne soient en relation avec leur dépôt
- Nous verrons comment on peut reconstruire ce répertoire .git plus tard
- Si votre projet est organisé en arborescence, vous ne verrez pas de répertoire .git dans chacun des sous-répertoires. Le répertoire .git est situé dans la racine de votre dépôt

Le contenu du répertoire .git

- **Branches :** n'est plus utilisé par les versions récentes de git
- **description :** est utilisé par GitWeb
- **info :** contient un fichier exclude permettant de spécifier les fichiers que l'on souhaite ne pas voir gérés par git (en général on préfère .gitignore)
- **hooks :** contient les scripts de procédures automatiques côté client ou serveur
- **objects :** base de données
- **refs :** stocke les pointeurs vers les objets commit
- **HEAD :** pointe sur la banche en cours
- **index:** l'endroit où Git stocke les informations sur la zone d'attente

Visualisation de la configuration

- dans le répertoire d'accueil d'un utilisateur le fichier *.gitconfig* contient les informations relatives à un utilisateur
- dans le répertoire du dépôt *.git/config* vous trouverez les informations relatives à votre dépôt
- Pour lister les paramètres en mode console

```
git config --list
```

```
user.email=gilles@gillesmaire.com
user.name=Gilles Maire
push.default=matching
core.autocrlf=input
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=ssh://gillesm@git.code.sf.net/p/dicorime/qdicorime
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

Configuration basique

- Informations utilisateur

```
git config --global user.name "Gilles Maire"
git config --global user.email gilles@gillesmaire.com
```

- Informations sur l'environnement

```
git config --global core.editor vi
git config --global color.diff auto
git config --global color.status auto
git config --global color.branch auto
```

Status et log

Status

- La commande la plus utile est `git status` : elle va indiquer les fichiers non gérés par git et ceux qui sont dans l'état staging. Elle n'affiche pas les fichiers commités.
- Sous un environnement avec interface graphique on ne peut pas la lancer sur un répertoire qui n'est pas géré par git alors que sous Unix on peut :

```
git status

fatal: Not a git repository (or any parent up to mount point /home)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
```

- Si un répertoire est géré par git on verra les fichiers qui sont non gérés, ceux qui sont en attente de validation.

```
git status

Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)
    nouveau fichier : fic2.txt
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
```

Plus d'options sur la commande status

- Afficher en format court

```
git status -s

A  fic2.txt
?? fic3.txt
```

- Affiche en format court mais sans caractères ANSI (pas de couleur, pas de gras)

```
git status --porcelain
```

- Les conventions sur les lettres affichées sont les suivantes : A (Ajouté), D (Détruit), R(Renommé), C(Copié), U(Updated)

Lister les fichiers d'un dépôt

- On peut lister les fichiers d'un dépôt en complément des commandes de log que nous verrons par la suite
- Cela s'obtient par la commande

```
git ls-tree --full-tree -r HEAD
```

- Avec les options suivantes :
  - `-r` : pour récursif
  - `-d` : n'affiche que les répertoire
  - `--full-tree` : affiche l'ensemble de l'arborescence même si vous êtes positionné dans un sous répertoire
  - `-l` : affichage en format long avec la taille du fichier

Historique des validations

- Afficher par ordre chronologique inversé l'ensemble des modifications qui ont été effectuées. `git help log` affiche les nombreuses options.

```
git log
git log fichier # montre les modification du fichier log
```

- Montrer les différences entre chaque modification à l'aide de la commande `diff`

```
git log -p
```

- ne montrer que 4 niveaux de log

```
git log -4
```

- Afficher le nombre de fichiers changés pour chaque modification

```
git log --stat
```

- Afficher chaque modification sur une seule ligne ou plusieurs

```
git log --pretty=oneline|short|full|fuller|format
```

Format retour git log

- La commande git log renvoie :
  - la date et l'auteur des changements
  - le commentaire entré dans le logiciel via l'éditeur ou l'option `-m` de la commande `git commit -m`
- un numéro de version sous la forme SHA1 :

```
8ce19d9926ab783c8af51ccd8d44d09aee89ccfa
```

- L'aide indique que l'on peut choisir un format de sortie après l'option `format` mêlant des variables %an (nom auteur) %s(sujet) %ar (date relative) %h(SHA1) et bien d'autres variables...

```
git log --pretty=format:"%h - %an, %ar : %s"
git log --graph : permet d'afficher les log avec un graphe
git log --name-status : affiche les fichier concernés dans chaque commit
```

Utilisatione la commande gitk

Démarrer un dépôt vide

Initialisation via init

- Démarrer un dépôt vierge :

```
mkdir repertoire ;cd repertoire
git init
```

- cela aura juste pour effet de créer un répertoire `.git` dans votre projet
- On peut voir qu'aucun fichier n'est suivi par git

```
git status
```

- On créé un fichier texte fic1.txt avec un éditeur dans lequel on ajoute un caractère

```
git status

Fichiers    non suivis:
fic1.txt
```

- On ajoute le fichier

```
git add fic1.txt
git status
```

L'option bare

- Quand on initie un dépôt dans lequel on ajoute des fichiers on utilise la commande vue précédemment à savoir

```
git init nomdepot
```

- Par contre si on veut que ce dépot soit un dépôt sur lequel personne n'entrera de commandes, il faut le déclarer sans répertoire de travail comme ceci :

```
git init --bare nomdedepot
```

La commande add

- Ajouter des fichiers qui ne sont pas pris en compte dans le dépôt git local ou qui ont été modifié

```
git add fichier
git stage fichier
```

- cette commande permet également d'indexer le fichier qui était dans le dépôt. Indexer veut dire qu'on calcule le checksum SHA d'un fichier afin de s'assurer de son identité avec celui du serveur.

```
# Ajouter tous les fichiers d'extension c du répertoire
git add repertoire/*.c

# Ajouter tous les fichiers
git add .
```

1. refaire un git add sur des fichiers déjà indexés ne pose pas de problème. Aussi on ne se prive pas d'exécuter un `git add *` sur un répertoire pour lequel on veut être sûr que tous les fichiers soient pris en compte.
2. faire plusieurs git add avant un commit est tout à fait possible, autrement dit on peut ne faire qu'un commit pour plusieurs modifications

Commit

Rubriques

- Commit
- Entre staging et commit
- Étiquettes

Commit

La commande commit

- valider un fichier en staging, sans option on devra renseigner un commentaire de modification dans un éditeur

```
git commit fichiers
```

- saisir le commentaire sans avoir à le rentrer dans l'éditeur pour le fichier désigné

```
git commit -m "le commentaire" fichier
```

- Pour placer tous les fichiers qui ont été modifiés en staging sans avoir à effectuer le add nécessaire pour la mise à jour :

```
git commit -am "le commentaire"
```

- **NB :** le commit n'envoie pas les modifications aux serveurs connectés contrairement à SVN, il ne sert qu'à mettre à jour le dépôt local.

Exercice init add et commit

- Dans cet exercice nous allons utiliser des petits poèmes japonais

d'une à trois lignes (Haïkus).

- Créer un répertoire Haikus
- Créer trois fichiers Haïkus dans ce répertoire
- Créer un dépôt git
- Commiter le premier Haïkus, mettre le deuxième en ajout et ne pas ajouter le troisième
- Exécuter `un git status` pour visualiser l'état de chacun des fichiers
- Modifier chacun de Haïkus
- Exécuter un `git status`

Pour les Unixiens

- La commande `git add` peut s'utiliser avec l'option -p qui montre le patch qui va être être appliqué au fichier et pose la question de la continuation

```
git add -p fic1.txt
```

```
git add -p fic1.txt
diff --git a/fic1.txt b/fic1.txt
index 7898192..68bf081 100644
--- a/fic1.txt
+++ b/fic1.txt
@@ -1 +1 @@
-a
+an
Stage this hunk [y,n,q,a,d,/,e,?]?
```

- l'appui sur ? explicite les choix possibles

```
y  : oui , n : non, q :quit,  all : tous à oui , d : aucun etc...
```

gitignore

- Pour éviter de voir des alertes concernant des fichiers non indexés (fichiers temporaires des éditeurs, fichiers exécutables etc) on peut mettre les patrons de ces fichiers dans le fichier `.gitignore` qui est situé dans la racine (et non pas dans le répertoire .git)
- **Exemple :**
  - `*.[oa]` (ignore les fichiers objet et les librairies statiques)
  - `*~` tous les fichiers d'extension ~
- Les lignes commençant par # dans le fichier .gitignore sont des commentaires
- une ligne finissant par / désigne un répertoire
- un ! en première colonne indique les fichiers à indexer malgré l'action d'autres règles présentes dans le fichier
- Enfin pour indexer un fichier même s'il est dans `.gitignore`

```
git add --force file
```

- Le fichier .gitignore peut être inclus ou ne pas être inclus dans les fichiers commités, en fonction de la présence de fichiers génériques ou propres à l'utilisateur.

gitignore

- Créer un fichier notesperso.txt à l'aide d'un éditeur
- Mettez le nom de ce fichier dans le fichier `.gitignore`
- Effectuer un git status

Entre staging et commit

Modification de document

- Prenons trois fichiers fic1.txt (commité) fic2.txt (staging) fic3.txt (non suivi)
- Modifions à l'aide d'un éditeur

```
git status -s

M fic1.txt
AM fic2.txt
?? fic3.txt
```

- On voit que les fichiers suivis sont toujours marqués modifiés (fic2 n'est plus AM)

```
git commit
git status -s

 M fic1.txt
 M fic2.txt
?? fic3.txt
```

- Un git add est nécessaire pour passer les fichiers en mode A puis un commit les passera en mode commité

Commande rm et mv

- En effaçant un fichier par une commande d'effacement de votre OS sans passer par `git`, le fichier sera marqué modifié mais non indexé, il faut passer par `git rm` fichier pour l'indexer
- on peut vouloir arrêter de suivre un fichier mais ne pas le détruire physiquement du disque :

```
git rm --cached fichier
```

- Commande mv

```
git mv fic1 fic2
```

- est équivalent à

```
mv fic1 fic2
git rm fic1
git add fic2
```

Annuler les actions

- Modifier le message de validation que le fichier ait été indexé ou pas :

```
git commit --amend
```

- Annuler un git commit c'est à dire passer du HEAD à la zone staging

```
git reset fichier
# ou git reset fichier HEAD comme donné par git status
```

- récupération de la dernière version en zone staging

```
git checkout fichier
```

- Annuler toutes les dernières modifications en staging (sans mentionner de fichier):

```
git checkout .
```

Quelques commandes utiles

- Vous avez détruit le fichier toto de votre répertoire de travail mais heureusement il était indexé

```
git checkout toto
```

- Vous avez détruit plusieurs fichiers qui étaient indexés dans votre répertoire de travail (et les fichiers modifiés non commités perdront leurs modifications)

```
git checkout .
```

- supprimer les fichiers / répertoires qui ne sont pas suivis par git

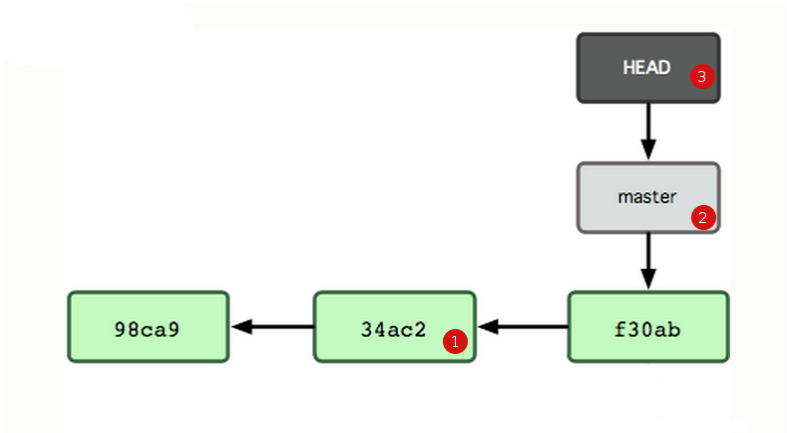
```
git clean -f
git clean -d
```

Les pointeurs HEAD et master

- Nous avons vu le mot **HEAD**, c'est un pointeur vers la révision courante. C'est en général la dernière révision mais ce pointeur peut être déplacé vers un autre emplacement que que la révision courante.
- Le mot **master** est également apparu, c'est le nom communément donné à la branche principale (nous allons voir que nous pouvons avoir plusieurs branches). Ce nom *master* n'est qu'une convention et pourrait être n'importe quel autre nom.
- On trouve aussi la notation HEAD~ qui signifie le parent de HEAD

HEAD et master

- La représentation est faite sous la forme suivante



1. Les SHA1 identifient de façon unique chacun des états
2. La branche master est la branche principale
3. Le pointeur HEAD est par défaut sur master mais on peut le changer d'emplacement

Étiquettes

Étiquette légère annotée signée

- L'étiquetage consiste à renommer un numéro SHA1 en un nom plus facile à retenir
- Il existe trois sortes d'étiquettes
  1. *annotée* par un commentaire (option -a et -m). Une étiquette annotée contient la date de création, le nom du créateur, la signature du tag...
  2. *signée* si l'étiquette est signée par une clé GPG qui doit être installée sur votre poste (option -s)
  3. *légère* sans commentaire et sans signature (ni -m, ni -s, ni -a)
- Création d'une étiquette :

```
git tag -a v1.4 -m "version 1.4"
git tag -s v1.5 -m "my signed 1.5 tag"
git tag v1.6
```

- Création d'une étiquette à postériori sur un numéro SHA1

```
git tag -a v1.2 -m "Version 1.2" a6b4
```

Opération sur les étiquettes

- Lister des étiquettes qui ont déjà été attribuées dans l'ordre alphabétique des *tags*:

```
git tag
```

- Vérifier si une étiquette à été signée GPG

```
git tag -v  version
```

- Supprimer une étiquette

```
git tag -d etiquette
```

Travailler avec une ancienne version

- Poser une étiquette sur une ancienne version
  - On récupère la SHA1 de la version par la commande

```
git log
```

```
commit 81ffd753077b8bb821b64ecb953956cc5c81e039
Author: Gilles Maire <gilles@gillesmaire.com>
```

- on pose l'étiquette pour éviter d'avoir à noter le SHA1

```
git tag -a V0.8 -m "Version 0.8" 81ffd753
git commit -am "Commit avant de faire un checkout"
```

- Récupérer une ancienne version du dépôt

```
git checkout "V0.8" #ou git checkout 81ffd753
```

Pousser les étiquettes sur le serveur

- Par défaut, la commande git push ne transfère pas les étiquettes vers les serveurs distants. Il faut explicitement pousser les étiquettes après les avoir créées localement. Ce processus s'apparente à pousser des branches distantes

```
git push origin version2.8
```

- Si vous gérez de nombreuses étiquettes que vous souhaitez pousser en une fois, vous pouvez aussi utiliser l'option –tags avec la commande git push. Ceci transférera toutes les nouvelles étiquettes vers le serveur distant.
- Par chacune de ces commandes les utilisateurs qui effectueront une commande pull récupèreront les étiquettes poussées.

Remote

Rubriques

- Installer Git sur un serveur
- Dépôt distant
- Conflit

Installer Git sur un serveur

Git sur un serveur

- On utilise un serveur qui accueillera le dépôt Git et qui sera accessible en permanence.
- On verra que comme personne n'a besoin de voir les fichiers sur le serveur, on ne reproduit pas l'architecture des fichiers mais uniquement la base de données.
- git peut utiliser quatre protocoles : local, ssh, Git et Http(s)
  - local : le serveur est installé dans un autre répertoire de votre ordinateur, cela peut être sur un serveur NFS accessible à un groupe de travail
  - ssh : git accède aux serveurs distants par ssh et scp, par contre l'accès anonyme est impossible dans ce mode.
  - git : souvent réservé en lecture seule car ce protocole n'est pas authentifié. Nécessite l'installation d'un démon git sur le port 9418 donc l'ouverture de pare feu.
  - https : simple à mettre en œuvre et facile à gérer, par contre ce n'est pas le plus rapide d'accès

Création du dépot

- On crée de préférence un dépôt nu (c'est à dire qui ne contient pas de copie du répertoire de travail) à partir d'un dépôt existant. Cela se fait la commande
- `git init --bare mon_project.git`
- Traditionnellement on ajoute un tel dépôt avec l'extension git. Vous aurez ainsi un répertoire `mon_projet.git`.
- *bare* veut dire nu en anglais.
- Les utilisateurs ayant un accès ssh sur le serveur pourront accéder au dépôt en lecture par la commande

```
git clone utilisateur@serveur:/chemin/mon_projet.git
```

- Pour optenir l'accès en écriture sur le serveur à l'intérieur du répertoire mon_projet.git vous entrez la commande avec l'option shared sous une des formes

```
git init --bare --shared
git init --bare --shared group
```

Accès au serveur git

- Pour une petite équipe, vous pouvez créer sur le serveur autant de comptes que nécessaire via la commande adduser sous Linux
- La plupart du temps, on créé un seul compte git et dans le répertoire /home/git/.ssh/authorized_keys on ajoute les clés publiques de chaque utilisateur
- Rappel : pour installer un serveur ssh sur un serveur ubuntu on a juste besoin d'entrer une commande

```
sudo apt-get install openssh-server
```

- Pour générer une clé publique sur chaque compte distant les utilisateurs doivent entrer la commande :

```
ssh-keygen
```

Git en accès public

- On peut installer un serveur Web dont la racine pointe sur le dépôt Git en activant une option dite post-update
- cd projet.git/.git
- mv hooks/post-update.sample hooks/post-update
- chmod a+x hooks/post-update
- (Après qu'on ait poussé (push) par ssh, le serveur met à jour la partie accès par http)
- Dans la partie Apache on ajoute un virtual host

```
<VirtualHost *:80>
    ServerName git.gitserveur
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

Dépôt distant

Travailler avec un dépôt distant

- Lister les dépôt distants, si un dépôt à été cloné renverra le mot origin :

```
git remote
```

- Donne l'url stockée pour chaque origine

```
git remote -v
```

- on peut renommer/supprimer un serveur distant en un autre :

```
git remote rename nom1 nom2
git remote rm nom
```

- Ajouter des dépôts distants avec un autre nom de dépôt :

```
git remote add [nomcourt] [url]
```

Exemple :

```
git remote add ancien git://github.com/gmaire/scid
```

Récupération via clone

- Pour récupérer les sources depuis un serveur Git distant au choix

```
# Pour créer un répertoire repertoire
git clone URLDepot repertoire
# créera le répertoire depot
git clone URLDepot
```

- Exemples :

```
 # création du répertoire monprojet
git clone git://github.com/schacon/grit.git monprojet
 # création du répertoire monrepertoire
git clone git://github.com/schacon/grit.git monrepertoire
 # création du répertoire depot (fin de l'arborescence)
git clone ssh://gilles@ignu.fr:/home/gilles/depot
 # création du répertoire mondepot
git clone ssh://gilles@ignu.fr:/home/gilles/depot mondepot
 # clonage d'un dépot local
git clone repertoire repertoire2
 # clonage via http
git clone http://gilles@ignu.fr:depot mondepot
```

Récupération des versions distantes

- Pour récupérer la dernière version depuis un dépot distant qui a été cloné au préalable

```
git pull
```

- Pour récupérer la dernière version depuis un dépôt. Ceci est utile si un projet a été cloné par un utilisateur, puis modifié et qu'on souhaite effectuer les modifications sur le serveur initial qui ne connaît pas le dépôt distant.

```
git pull CheminDuDepot
```

- On aura tendance à ajouter sur le dépôt initial le deuxième dépôt :

```
git remote add CheminDuDepot
```

Soumission des modifications locales à un dépôt distant

- Ceci nécessite que vous vous identifiez vers le serveur distant, même les projets open source nécessitent que vous soyez identifié pour participer à son développement.
- On soumet le HEAD vers un dépôt distant via la commande

```
git push URL branche
git push origin master
```

- *origin* est le nom distant
- *master* est la branche courante
- Si le dépôt local à été cloné *origin* et *master* sont positionnés et on peut se contenter de la commande

```
git push
```

Séquence de travail normale avec dépôt distant

- La sequence de travail normal est la suivante :
  - modification des fichiers locaux
  - git add (sur les fichiers nouveaux)
  - git commit -am “messages de modification”
  - git pull (suivi de résolutions de conflits)
  - git push
- Un conflit peut être
  - automatiquement résolu par un merge fait par le pull
  - résolu manuellement dans le fichier fautif ou les différentes sont marquées par les caractères <<<< (modif locales) ==== et >>>> (modif distant)

Clone

- Créer un dépôt
- Le cloner
- Ajouter vos fichiers Haikus à ce dépôt

Conflit

Conflit entre plusieurs utilisateurs

- Robert modifie un fichier appelé config.cpp
- Il fait un 'git commit -am "modif bug cocacola"
- Il fait d'autres modifications
- Pendant que Kevin modifie le même fichier avec des choses différentes ET incompatibles avec la version de Robert
- Kevin fait un 'git commit -am "correction bug cc"
- Puis Kevin fait un `git push`
- En fin de journée Robert fait un `git push` : ce git push lui est refusé car des choses ont changé sur le dépôt on lui demande de faire un git pull
- Robert fait un `git pull` et il voit un message lui indiquant le conflit avec le message

```
CONFLIT (contenu) : Conflit de fusion dans config.pp
```

- Robert ouvre le fichier enlève les lignes incompatibles et fait un git commit -am "résolution conflit avec Kevin"
- Robert fait un git push qui cette fois-ci passe.

Les symboles de conflit dans les fichiers

- En cas de conflit le message suivant est renvoyé par git :

```
Fusion automatique de fichier1
CONFLIT (contenu) : Conflit de fusion dans fichier2
La fusion automatique a échoué ; réglez les conflits et validez le résultat.
```

- Le fichier `fichier2` contient alors les causes de conflit sous la forme

```
<<<<<<< HEAD
- deux
=======
- un
>>>>>>> branche
```

Les conflits de fusion

- Il peut arriver que la fusion soit impossible à réaliser sans intervention manuelle.
- Ce sera le cas si chaque branche apporte des modifications différentes au même endroit dans le même fichier.
- On dit qu'il y a conflit.
- Dans ce cas là, Git interrompt le *merge* et insère des marqueurs dans les fichiers conflictuels.
- Il faut éditer les ou les fichiers manuellement ou à l'aide d'une interface spécifique (git mergetool, ou meld)
- On peut voir les conflit à l'aide de la commande `git status`

Conflit

- La séquence suivante montre les incohérences entre la première partie (mes modifications) et la partie du bas : les données du fichier

```
<<<<<<< HEAD:index.html

<body id="home" lang="en">

=======

<body id="home" lang="fr">

>>>>>>> dev:index.html
```

- Il faut choisir la portion de code que l'on garde, qui peut être un panachage des lignes et supprimer les lignes <<< >>> et ===

Résolution du conflit

- On supprime les symboles <<<<<<<, >>>>>>> et ======= et les lignes erronnées comprises entre ces symboles
- On fait au choix :

```
git add git.mmd
git commit -m "conflit résolu"
```

ou

```
git commit -am "conflit résolu"
```

ou une combinaison des lignes justes entre les premières et les dernières

- On pousse les modifications (si on travaille avec un serveur distant)

```
git push
```

Branches

Rubriques

- Branches
- Merge
- Particularismes
- Rebaser

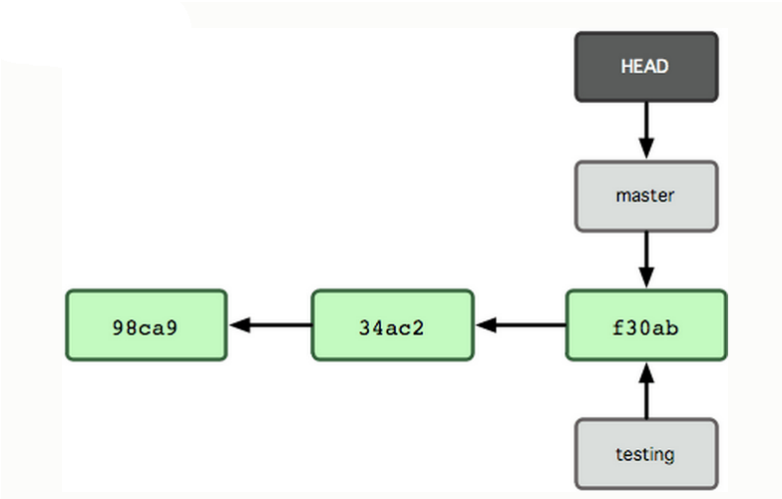
Branches

Définitions

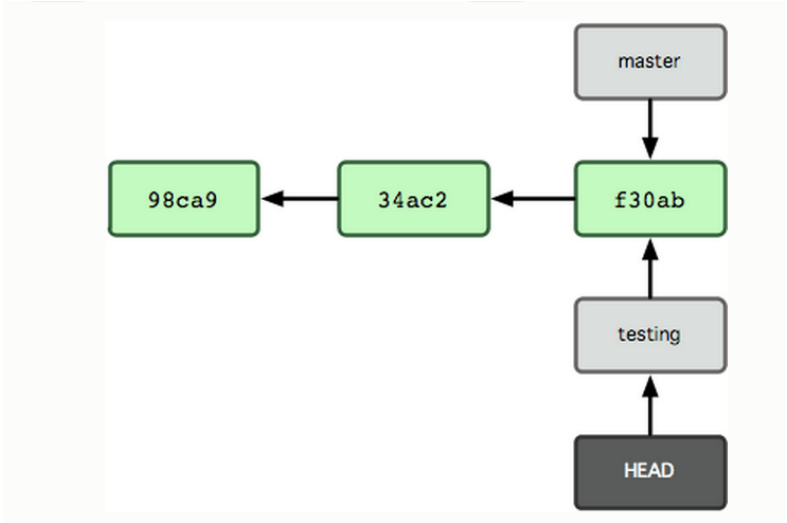
- Créer une branche signifie diverger de la ligne principale de développement et travailler en dehors de cette ligne principale
- Les autre VCS quand ils font des branches, recopient récursivement les répertoires ce qui est très lent.
- Git lui fait des photos instantanées et encourage l'utilisation des branches et de fusions (merge) plusieurs fois par jour.
- Une branche est une étiquette qui pointe vers un commit
- La branche principale créée par défaut s'appelle la branche *master*

Principe branch/checkout

git branch testing

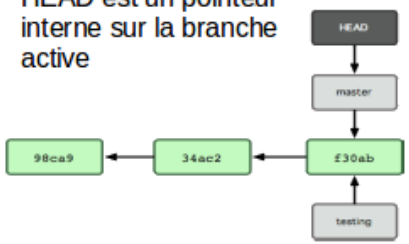


git checkout testing

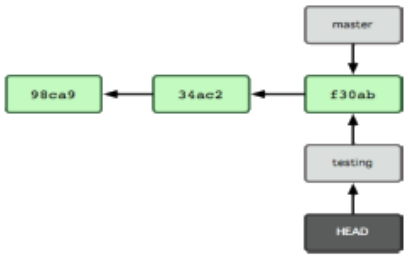


Principe de Branche après commit

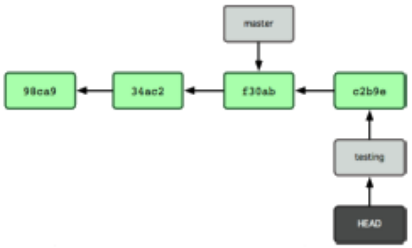
HEAD est un pointeur interne sur la branche active



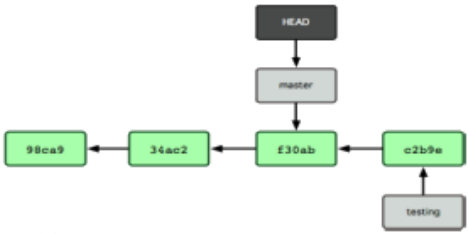
git branch testing



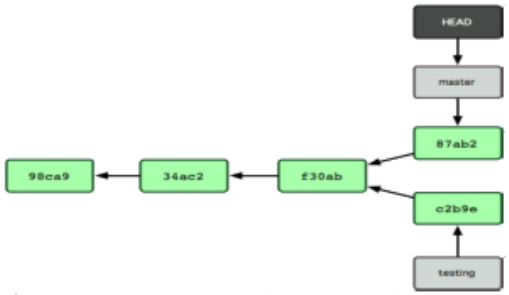
git checkout testing



git commit -a -m 'modif1'



git checkout master



git commit -a -m 'modif2'

Remarque :
git branch test
git checkout test
<=>
git checkout -b test

Commandes

- Créer d'une branche :

```
git branch branche
```

- Créer une nouvelle branche nommée «branche» et passer dessus

```
git checkout -b branche
```

- Retourner sur la branche principale

```
git checkout master
```

- Supprimer la branche «branche»

```
git branch -d branche
```

- Créer une branche sur un SHA

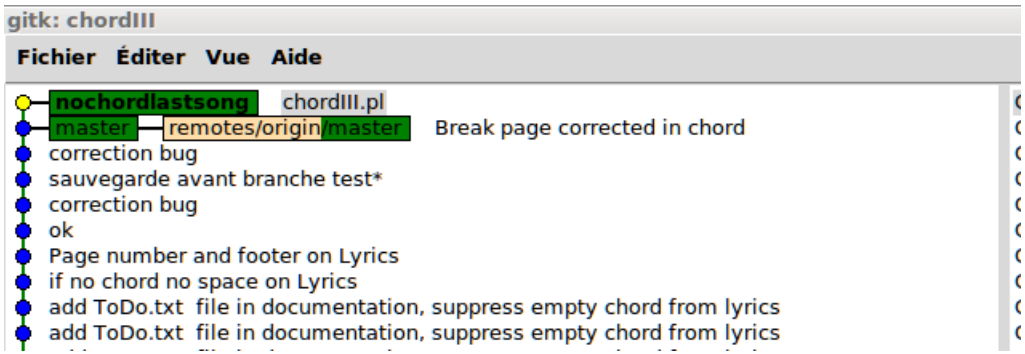
```
git branch branche SHA
```

Visualiser la branche active

- En mode commande :

```
git branch
git branch -v
git branch -vv
```

- Avec gitk la branche active est de couleur jaune



Création de Branche

- Créer une branche sur votre dépôt contenant les Haïkus
- Dans cette branche ajouter une date de publication en bas de chaque Haïkus
- Revenez sur la branche master et ajouter un titre sur chaque Haikus
- Passer de la branche principale à votre branche pour constater que votre fichier change de contenu

Merge

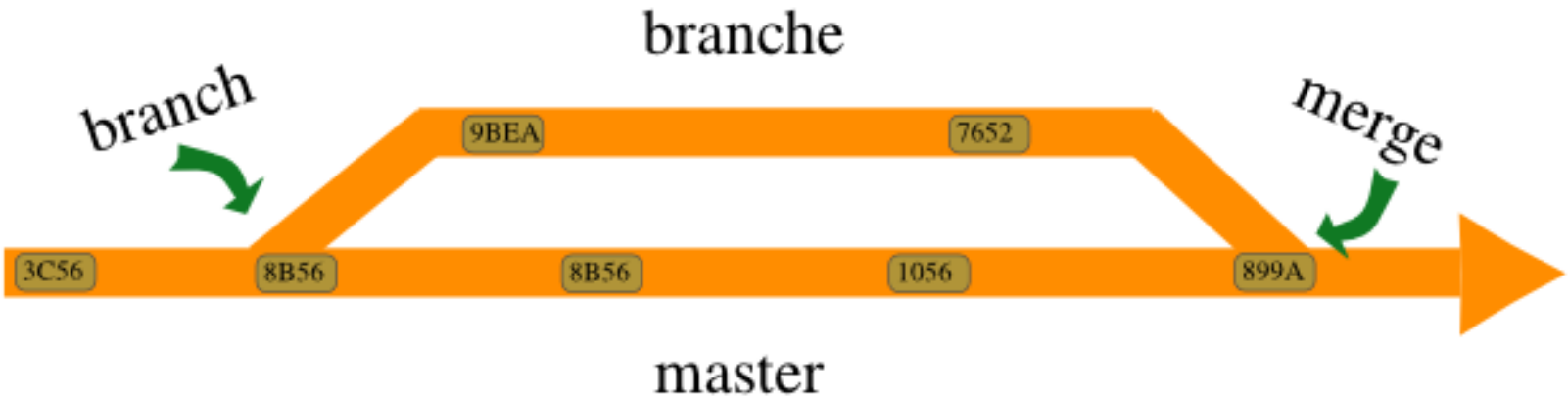
Merge

- On peut effectuer un *merge* pour fusionner la branche avec le projet *master* ou avec une autre branche par la commande :

```
git merge nom_branche
```

- Remarque : il faut être sur la branche master pour faire le merge :

```
git checkout master
```



Résolution de conflit sur un merge

- On se positionne sur la branche master

```
git checkout master
```

- On supprime les symboles <<<<<<<, >>>>>>> et ======= et les lignes erronnées comprises entre ces symboles
- On commite le conflit

```
git add git.mmd
git commit -m "conflit résolu"
```

- On supprime la branche

```
git branch -d nombranche
```

Fetch

- Nous avons vu comment ajouter un dépôt via la commande

```
git remote add
```

- Ceci est assez utile pour ajouter le dépôt d'un autre développeur par exemple pierre:

```
git remote add pierre ssh://git@ignu.fr/developpement/dev.git
```

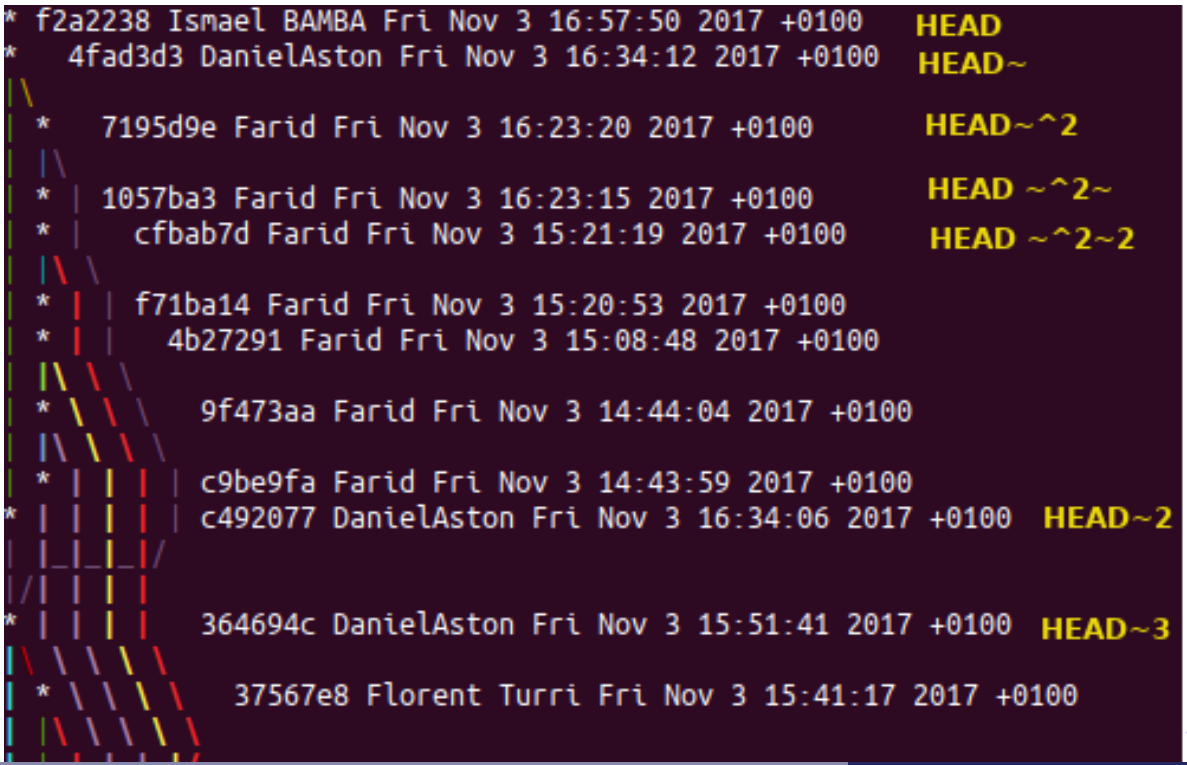
- un git remote -v montrerait la liste des dépots sous la forme

```
origin  ssh://gillesm@git.code.sf.net/p/guitarrosette/git (fetch)
origin  ssh://gillesm@git.code.sf.net/p/guitarrosette/git (push)
pierre  ssh://git@ignu.fr/developpement/dev.git (fetch)
pierre  ssh://git@ignu.fr/developpement/dev.git (push)
```

- La commande `git fetch pierre` permet récupérer l'ensemble des données depuis le dépôt pierre, mais ces données seront créées sous forme d'une branche
- La commande `git pull pierre` à la même action mais elle ajoute un `git merge` que nous verrons dans la gestion des branches

HEAD^ et HEAD~

- HEAD~ est le parent de HEAD, HEAD~2 est le parent de HEAD~ ,- HEAD~3 est le parent de HEAD~2
- HEAD^2 est le deuxième parent de HEAD
- HEAD~^2 est le deuxième parent de HEAD~
- HEAD~1 est équivalent HEAD~ et à HEAD^1(premier parent)



Remarques sur les HEAD

- **Attention :** pour visualiser les deuxièmes parents gitk est moins fiable que la commande git log –graph
- On peut utiliser les notation HEAD~ et HEAD^ dans les commandes git classiques par exemple pour créer la branche toto au parent concerné

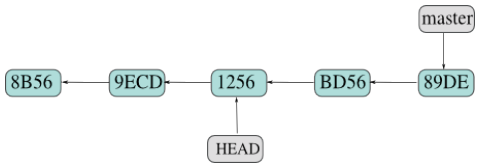
```
git branch toto HEAD~^2~3
```

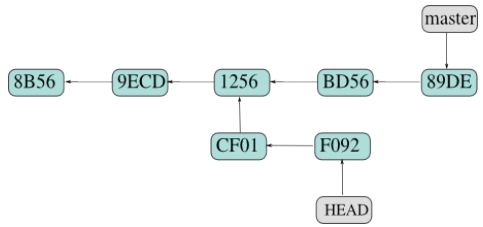
Exemple d'utilsation des HEAD

Particularismes

Branche détachée

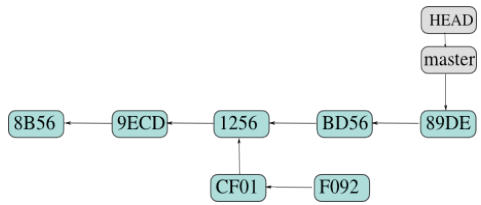
```
git checkout 1256
git commit -a -m v1
git commit -a -m v2
```





- la commande git checkout master va signaler qu'on perd la branche

```
git checkout master
Attention : vous abandonnez 1 commit, non connecté à une branche :
  b099891 ok
```



- La commande `git log` ne montrera pas les commit de la branche sans nom. On peut les retrouver par la commande `git reflog` qui donne la liste des déplacements de HEAD, on peut alors poser une branche sur le SHA1 ou appeler

Liste des branches

- Pour afficher la liste des branches (la branche master sera marqué un *)

```
git branch
```

- Pour afficher le HEAD et les branches

```
git log --graph --oneline --decorate
```

- Pour afficher la liste des branche qui ne sont pas fusionnées

```
git branch --no-merged
```

- Pour afficher la liste des branches qui sont fusionnées

```
git branch --merged
```

Supprimer une branche

- Supprimer une branche qui a été fusionnée se fait par la commande

```
git branch -d branche
```

- Si on effectue la même commande sur une branche qui n'a pas été fusionnée nous recevrons un message d'erreur

```
error: La branche 'Nouvelle' n'est pas totalement fusionnée.
Si vous êtes sur que vous voulez la supprimer, lancez 'git branch -D Nouvelle'.
```

- Dans ce cas on utilise la commande

```
git branch -D branche
```

Pousser les branches

- Une branche peut être locale quand elle permet de basculer d'un espace de travail de corrections de bug à un espace de travail d'ajout de fonctionnalités par exemple
- Elle peut également être globale à tous les intervenants d'un projet : la branche version en cours, la branche release candidate etc..
- Les branches locales ne sont pas transmises par défaut au serveur distant lorsque vous soumettez votre travail par une commande `git push`. Cela revient à dire que les branches par défaut sont locales à un dépôt
- Si on souhaite rendre la branche publique nous devons ajouter les options de branche au push :

```
git push NomServeur NomBranche
git push origin NomBranche
```

- On peut spécifier un nom différent, celui que prendra la branche distante

```
git push origin NomBranche:NomBranchePublique
```

- La branche une fois poussée pourra être récupérée avec un clone, un pull ou un fetch mais le nom des branches aparaîtra non pas sous le nom `master` mais `remotes/origin master`

Branches existantes

- Créer un dépot
- Le cloner
- Créer dans le clone des fichiers et une branche
- Pousser les fichiers et la branche vers le dépot distant
- Recloner le dépot

Travail sur une branche distante

- Pour récupérer les nouvelles versions de la branche origin/master nous pouvons procéder à un

```
git fetch origin
```

- Pour travailler sur cette branche nous devons lancer un merge sur notre copie de travail

```
git merge
```

- Mais on peut préférer créer une branche locale contenant* cette branche :

```
git checkout -b correctionserveur origin/correctionserveur
```

- On peut aussi vouloir cloner un dépot avec uniquement une branche

```
git clone --single-branch --branch branche urldepot
```

Effacer une branche distante

- L'effacement d'une branche distante devenue inutile après avoir subi les merges sur le master se fait par :

```
git push origin :NomBranche
```

- C'est une commande compliquée à retenir mais en fait c'est la même syntaxe que la commande push

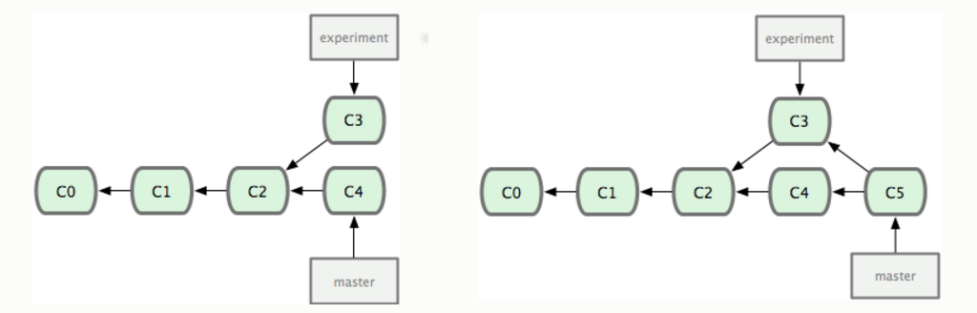
```
git push origin NomBrancheLocale:NomBrancheDistante
```

- comme NomBrancheLocale est vide cela revient à dire on met à vide la branche distance NomBrancheDistante

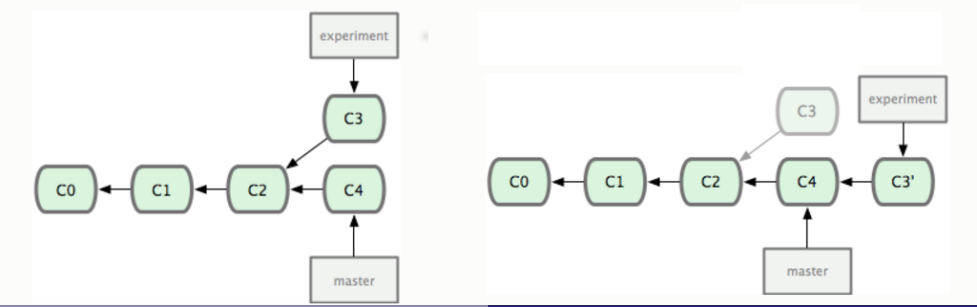
Rebaser

Rebaser

```
git checkout master
git merge experiment
```




```
git checkout experiment
git rebase  master
git checkout master
git merge experiment
```



Conflit sur les rebase

- La commande rebase ne gardant pas les branches en cas de conflit, vous serez inviter à corriger le conflit et à exécuter non pas un commit qui n'aurait aucun sens mais un

```
git rebase --continue
```

- Il faut, avant de continuer aller consulter dans le répertoire .git/rebase-apply/patch qui indique les problèmes rencontrés pour récupérer les noms des fichiers coupables
- on peut alors modifier les fichiers coupables dans le répertoire de travail ne pas oublier de faire sur chaque fichier coupable un

```
git add fichier
```

- On a aussi la possibilité de revenir en arrière via une commande

```
git rebase --abort
```

Les dangers du rebase

- La façon de résoudre les conflits est un peu moins intuitive et surtout on corrige non plus les fichiers du master mais les fichiers de la branche modifiée
- L'autre problème vient du futur push de votre travail sur le dépot distant. En effet vous allez envoyer sur la branche master le nouveau commit auquel il va manquer des SHA1 de la branche principale
- Si un autre développeur a posé une branche sur ce SHA1, il ne va plus le retrouver et se retrouver avec une branche détachée.
- En conclusion : **ne rebasez jamais des commits qui ont déjà été poussés sur un dépôt public.**

Remiser

- Lorsque vous travaillez sur une partie de votre projet, les choses sont dans un état instable
- Vous voulez peut-être travailler sur une autre branche sans pour autant valider et commiter des choses qui ne fonctionnent pas encore
- La solution est de remiser le travail sur la branche en cours via une commande

```
git stash
```

- Une fois que la commande a été exécutée elle est empilée sur la pile HEAD, on peut donc faire plusieurs stashs sur plusieurs branches pour retrouver l'ensemble des stashs

```
git stash list
```

```
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

- Pour retrouver le working directory dans l'état d'une remise

```
git stash apply  # pour le dernier
```

Supprimer une remise

- La dernière remise est appliquée et supprimée via la commande (idem apply + suppprimée)

```
git stash pop
```

- Suppression sans appliquer

```
git stash drop @{1}
```

- Voir les modifications appliquées par un stash

```
git stash show
git stash show -p  ( plus de détail)
```

Créer une branche à partir d'une remise

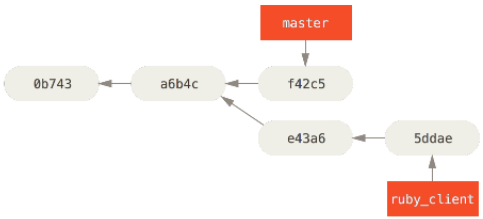
- Si votre remise est trop ancienne vous risquez de voir des conflits de fusion avec des fichiers qui ont été modifiés
- Vous aurez donc intérêt à créer une branche à partir d'une remise comme suit

```
git stash branch testchanges
```

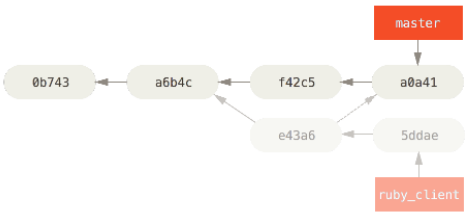
- Vous pourrez alors résoudre les conflits sur les fusions de branche
- Il faut savoir qu'on ne peut pas résoudre des conflits sur les remises et qu'on ne peut pas annuler le travail effectué par une remise de façon simple

Cherry Pick

- La commande cherry-pick permet d'appliquer un commit sur une autre branche. Ce picorage consiste à repérer une empreinte SHA1 sur la quelle on a fait un traitement, d'en extraire le patch et d'appliquer ce patch sur une autre branche.




```
git cherry-pick  e43a6fd3e94888d76779ad79fb568ed180e5fcdf
```



Rq : le SHA1 e43a6 a changé à cause du changement d'heure.

Méthodes

Rubriques

- Éclaircissements
- Organisation et astuces
- Personnalisation
- Le cas des fichiers binaires

Éclaircissements

La commande reset

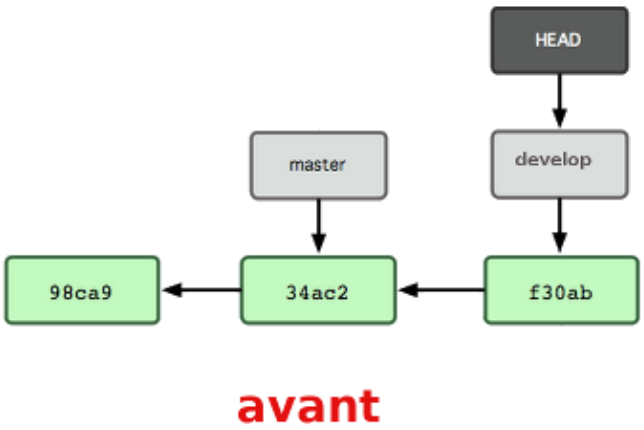
- La commande reset annule un commit en :
  - déplaçant le HEAD sur le commit précédent
  - mettant à jour l'index avec le contenu pointé par HEAD (fichier en M sur git status)
  - mettant à jour la copie de travail en phase avec l'index (contenu du fichier changé)
- Ainsi la commande reset dont la syntaxe est :

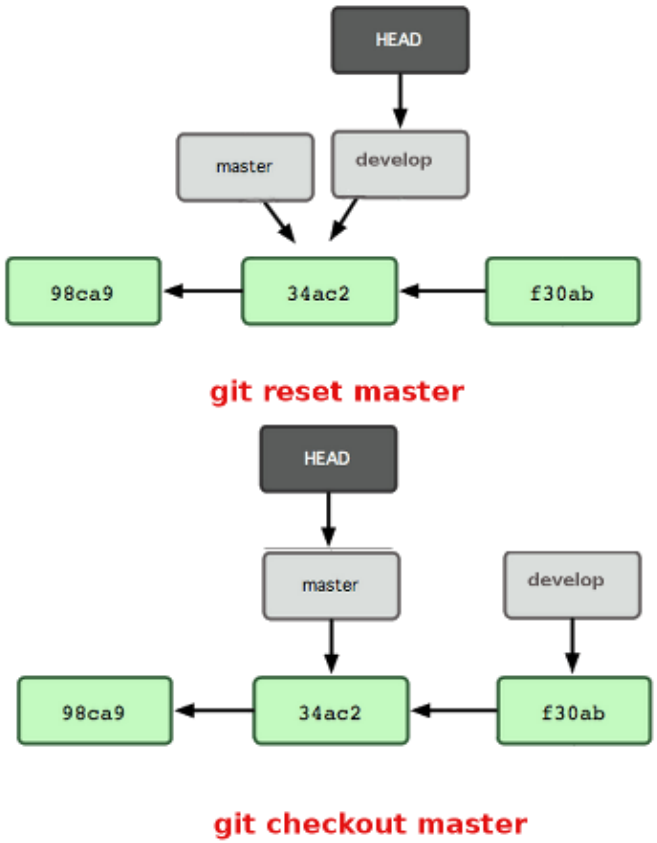
```
git reset HEAD~
#HEAD~ est le parent de HEAD
```

- Suivant les trois phase décrites précédemment la commande est équivalente à

```
git reset --soft HEAD~
git reset --mixed HEAD~
git reset --hard HEAD~
```

Différence entre checkout et reset





Le danger de reset

- La commande reset est jugée dangereuse par sa commande –hard (ou la commande toute en un) car c'est est un des très rares cas où Git va réellement détruire de la donnée.
- Toute autre invocation de reset peut être défaite, mais l'option –hard ne le permet pas, car elle force l'écrasement des fichiers dans le répertoire de travail.
- La commande reset peut être interdite. Par exemple gitolite permet que l'option W+ (rewind) ne soit autorisé qu'aux administrateurs

Git revert

- La commande revert permet de revenir sur un commit comme la commande reset, mais cette fois en préservant le précédent commit afin de permettre de garder la trace de ce commit et de ne rien détruire de manière irreversible.
- On verra donc dans les log le SHA du commit que l'on veut supprimer suivi de modification annulant ce commit
- Sa syntaxe est la suivante :

```
git revert HEAD~
```

ou

```
git revert SHA
```

Checkout

- La commande checkout est utilisée dans beaucoup de cas avec des usages différents :

```
git checkout    #  Sans argument : indique les status des fichiers non commités
git checkout Makefile    #  annule le staging du fichier ( c'est à dire annule
git checkout SVG           #  si SVG est un répertoire annule le staging sur le
git checkout  branche    #  passe le HEAD sur le pointeur de branche
git checkout tag         #   passe le HEAD sur le tag
git checkout SHA       #   passe le HEAD sur le SHA
git checkout SHA file # met le fichier dans l'état qu'il avait au commit SHA
```

- En cas de d'identité de nom de tag, de fichier ou de répertoire c'est la branche qui prime
- En cas d'identité de nom de tag ou de fichier : un message demandera de préciser

blame

- La commande git blame nom de fichier permet de visualser les différents intervenant ayant modifié un fichier :

```
git blame sha1_file.c
```

```
0fcfd160 (Linus Torvalds  2005-04-18 13:04:43 -0700    8)  */
0fcfd160 (Linus Torvalds  2005-04-18 13:04:43 -0700    9) #include "cache.h"
1f688557 (Junio C Hamano  2005-06-27 03:35:33 -0700   10) #include "delta.h"
```

Organisation et astuces

Taille des équipes

- Pour les petites équipes on se basera sur un modèle similaire à SVN avec un serveur de dépôt distant sur lequel chacun aura le droit de pusher
- Pour les projet importants, on peut donner accès en écriture à un dépot public par développeur et un accès en lecture seule pour chacun à un dépôt de référence. Dans ce cas on envoie une demande à l'intégrateur du projet pour qu'il fasse un pull depuis le serveur concerné pour contrôler la bonne intégration.
- On peut également ne donner l'accès au master en écriture qu'à certaines personnes, les développeurs n'ayant le droit de pusher que dans des branches.
- Pour les projets de taille pharaonique comme le noyau linux, les développeurs travaillent sur leur dépot dans un sous projet et des lieutenants récupèrent les développement sur leur dépot et une fois le travail vérifié, envoient une demande au responsable principal du projet qui se charge de l'intégration.

Remarque SVN Git

- Git à la différene de SVN qui résoud les conflit lors d'une soumission de code, refuse le push si le HEAD du dépôt distant est avant le HEAD de votre commit. En d'autres termes , si vous devez effectuer un pull pour être à jour, vous ne pourrez pas faire de push
- Ceci conduit à ne pas faire systématiquement un push avant de quitter le bureau, mais plutôt des push réguliers.

Interdire la branche master

- Les hooks sont des programmes qui seront exécutés par le serveur soit avant soit après une action
- Dans le répertoire .git/hooks de votre dépôt, vous verrez ces fichiers à l'état d'exemple.
- Changer le nom d'un de ces hooks.sample sans l'extention .sample
- La commande BRANCH=`git rev-parse --abbrev-ref HEAD`
- La variable $remote qui est le premier argument peut être testée pour vérifier qu'elle correspond à l'utilisateur responsable

Types de commit

- Chaque commit doit être homogène c'est à dire ne pas concerner quelques bouts de fonctionnalités mais une fonctionnalité simple.
- L'idéal est que chaque commit concerne une tâche n'excédant pas 30 mn de travail. On comprend qu'un commit représentant un mois de travail provoquera des désagréments chez les autres développeurs.
- Il est préférable de faire une branche que l'on merge et que l'on détruit avant un commit, au cas où un évènement viendrait à demander une action de votre part sur le code
- Il est bon également qu'un commit représente la résolution d'un ticket d'incident ou d'un bug, pas de quatre bug ou tickets d'incident.
- Si on a corrigé un autre bug qui n'avait pas de rapport avec le bug sur lequel on travaille, il est intéressant de ne pas le mettre en staging tout de suite, mais effectuer un commit avant de la passer en staging

Merge vs rebase

- Rebase est un bon outil quand on a mis en oeuvre une petite fonctionnalité, qu'elle fonctionne après quelques commit et qu'on ne souhaite plus conserver les commits.
- Ainsi l'historique ne conservera pas les petites modifications mais une fonctionnalité globale qui doit rester limitée pour ne pas gêner les autres développeurs

Complétions et alias

- Si vous avez installé une version de git correctement configurée, vous pouvez entrer `git` suivi de la touche `tab` pour disposer les completions.
- La complétion fonctionne pour les commandes complètes, mais également pour les fins de verbes ou pour les options.
- Vous pouvez définir des alias de commande par la syntaxe suivante :

```
git config --global alias.co commit
```

- L'emploi du global s'il est omis provoquera l'alias que sur le dépôt courant et non pas sur tous les dépôts

git lg

- Si on n'utilise pas d'interface graphique ou le visualisateur gitk, on ajoute un alias appelé lg via la commande :

```
git config --global alias.lg "log --color --graph --pretty=format:\
'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) \
%C(bold blue)<%an>%Creset' --abbrev-commit"
```

- la commande alias.lg permet de définir l'alias
- l'option –graph permet de dessiner un graphe
- l'option pretty-format et l'option %C permettent de définir les couleurs d'affichage

```
* 95db058 - (HEAD -> master, origin/master) ok (il y a 23 heures) <Gilles Maire>
* 712f5f6 - Décalage ligne accords (il y a 23 heures) <Gilles Maire>
* 7c22283 - todo modif (il y a 2 jours) <Gilles Maire>
*   4e4ac97 - Merge  annotation (il y a 2 jours) <Gilles Maire>
|\
| * 9b2e579 - merge and pagepos (il y a 2 jours) <Gilles Maire>
| *   defa8fe - merge pagepos and link (il y a 2 jours) <Gilles Maire>
| |\
| | * da432d1 - pagepos improvment in lyrics (il y a 2 jours) <Gilles Maire>
| | * 97ec57f - ok (il y a 2 jours) <Gilles Maire>
| * | 0ce7390 - modif (il y a 2 jours) <Gilles Maire>
| |/
* | 7684176 - Todo (il y a 2 jours) <Gilles Maire>
|/
```

Personnalisation

Configuration

- Le fichier /etc/gitconfig (sous linux) contient les configurations de tous les utilisateurs pour tous les projets git (ce fichier réagit à la commande git config –system)
- Ce fichier peut être complété par un fichier .gitconfig dans le répertoire d'accueil d'un utilisateur (ce fichier réagit à la commande git config –global)
- Si aucun de ces fichiers n'est renseigné, les valeurs utilisées seront lues dans le fichier .git/config de chaque projet git
- Si un fichier est présent dans le niveau local, il est prioritaire sur le niveau global lui même prioritaire sur le niveau system
- Pour visualiser l'ensemble des fichiers de configuration on ajoute l'option –show-origin à la commande config comme suit :

```
git config --list --show-origin
```

Les variables des fichiers config

- Ces variables se définissent suivant l'exemple suivant (ici donné avec global):

```
git config --global user.name "Gilles Maire"
```

- Voici les principales variables :
  - user.name : nom de l'utilsateur de git
  - user.email : adresse de l'utilisateur
  - core.editor : editeur de texte
  - commit.template : renseigne le nom du fichier template à proposer pour chaque commit
  - core.pager : nom du programme de pagination ou le désactiver en renseignant une ligne vide (")
  - user.signingkey : la signature GPG si vous signez vos commit
  - core.exludesfiles : spécifie des fichiers ignore supplémentaires au fichier que .gitignore
  - core.autocrlf : positionné à **true** remplace les LF en CRLF pour les utilisateurs de Windows, positionné à **input** pour que le correction soit faite uniquement en entrée
  - core.whitespace : - trailing-space (activé par défaut) : détecte les espaces en fin de ligne - before-tab (activé par défaut) : recherche les espaces devant les tabulations - indent-with-non-tab (non activé par défaut) : recherche les lignes qui commencent par 8 espaces plutôt que des tabulations - cr-at-eol (non activé par défaut) : accepte les retour chariot en fin de ligne

Différences entre le mode ligne de commande et fichier de configuration

- Si on ne veut pas entrer les commandes suivantes :

```
git config --global merge.tool extMergegit config
git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
git config --global mergetool.trustExitCode false
git config --global diff.external extDiff
```

- Le fichier .gitconfig sera de la forme

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
  trustExitCode = false
[diff]
  external = extDiff~
```

Configuration de l'outil de résolution des conflits

- Il existe plusieurs outils graphiques de résolution de conflit différents de la commande diff et indépendants des outils graphiques intégrés (donc principalement sous Linux)
- Citons : P4merge, Beyond compare, smartgit, kdiff et Meld.
- Pour configurer l'outil à afficher plusieurs paramètres de configuration sont nécessaires :
  - merge.tool : indique le programme à utiliser
  - mergetool.*.cmd : spécifie comment lancer la commande
  - mergetool.trustExitCode : indique le code de réussite de la fusion ou non
  - diff.external : indique la commande à lancer pour voir les différences
- On entrera donc les commandes suivantes par exemple pour meld:

```
git config --global merge.tool meld
git config --global mergetool.meld.cmd  'meld  \"$BASE\" \"$LOCAL\"
\"$REMOTE\" \"$MERGED\"'
git config --global mergetool.trustExitCode false
git config --global diff.external meld
```

Remarques pour les outils de merge

- Certains utilitaires de merge qui servent également d'utilitaire de diff, demandent 7 arguments en entrée. Or diff ne demande que deux arguments
- Il faut donc écrire un shellscript (ou un fichier bat sous windows) bâti sur le principe suivant pour le fichier appelé commande-diff :

```
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/commandediff "$2" "$5"
```

- L'outil de commandediff prendra alors la place de :

```
git config --global diff.external commandediff
```

Le cas des fichiers binaires

Problématique

- Les fichiers binaires ne sont pas les bienvenus dans git car :
  - git ne saura pas présenter les différences entre les deux fichiers binaires car la commande diff présente les différences entre deux fichiers ASCII
  - On se retrouve alors devant une problématique tout ou rien : soit on accepte le nouveau ficher binaire, soit on accepte l'ancien.
- Il est donc nécessaire de fournir non pas les exécutables mais les sources ayant servi à l'exécutable, non pas une base de données mais un dump permettant de restituer la base de données.
- Nous détaillerons deux méthodes :
  - l'une permettant de visualiser les différences entre deux fichiers binaires
  - l'autre permettant de visualiser les différences et de sauvegarder les fichiers binaires en fichier texte.

.gitattributes

- Nous allons ici traiter des diff de fichier Sqlite3
- le fichier .gitattributes contiendra une ligne de la forme :

```
*.db diff=sqlite3
```

- Ceci veut dire que pour tous les fichiers base de données d'extention db, le diff se fera en utilisant le driver sqlite3
- Nous allons écrire un driver sqlite3 simplement par la commande

```
git config diff.sqlite3.textconv  dumpsqlite3
```

- Le fichier dumpsqlite3 placé dans le path sera :

```
#!/bin/sh
sqlite3 $1 .dump
```

Utilisation des hooks

- Prenons le cas d'une base de données sqlite intégrée à un projet
- Cette base de données binaire si elle est changée souvent ne sera pas pratique car git indiquera des conflits ou des changements uniquement en signalant que le fichier a té modifié
- On peut utiliser un mécanisme de hook qui sont des programmes que l'on peut lancer avant une commande ou après une commande.
- Dans notre cas on va mettre en oeuvre deux programmes :
  - un pre-commit qui convertira avant chaque commit la base binaire en un dump texte qui sera sauvegardé sur le serveur
  - un post-merge qui reconvertira en binaire le fichier texte
- Ainsi des modifications de la base seront transformés en ordres SQL
- Par défaut les programmes hooks sont situés dans le répertoire .git/hooks qui n'est pas sauvegardé dans le git.
- Depuis la version 2.9, git permet d'utiliser des hooks situés par exemple dans le répertoire .githook du répertoire d'accueil
- On indiquera que les hooks sontà prendre dans ce fichier par la commande

```
git config core.hooksPath .githooks
```

Écriture des hooks

Plomberie

Rubriques

- Présentation
- Les arbres
- Références
- Opérations courantes

Présentation

Plomberie et porcelaine

- Dans je jargon Git, les commandes de bases (add, commit, pull, push, stash, rebase etc) sont appelées les commandes porcelaine.
- Celles concernant le fonctionnement interne de Git sont appelées plomberie, c'est à dire des commandes de bas niveau permettant d'effectuer des tâches plus complexes et relatives à l'organisation interne de Git.
- La compréhention de cette organisation interne commence l'étude du répertoire .git contenant une arborescence
  - des fichiers : config, description, HEAD, index, packed-refs
  - des répertoires : branches, hoioks, info, logs, objects, refs

Les fichiers

- Le fichier config contient les options de configuration spécifiques à votre projet
- Le fichier info contient un fichier d'exclusions listant les motifs que vous souhaitez ignorer et que vous ne voulez pas mettre dans un fichier .gitignore
- Le répertoire hooks contient les scripts de procédures automatiques côté client ou serveur
- Le fichier HEAD pointe sur la branche qui est en cours dans votre répertoire de travail
- Le fichier index est l'endroit où Git stocke les informations sur la zone d'attente
- Le répertoire objects stocke le contenu de votre base de données
- Le répertoire refs stocke les pointeurs vers les objets commit de ces données (branches),

Stockage des informations

- Chaque donnée est stockée sous la forme d'un couple clé/valeur
- La commande hash-object crée un couple clé/valeur stocké dans le répertoire objects (en fait dans un sous répertoire comprenant les deux premières lettres des hash)

```
echo 'test content' | git hash-object -w --stdin
```

- Chaque hash est codé sur 40 caractères, les deux premiers sont regroupés par répertoire, les 38 autres servent de nom de fichier.
- Le contenu de chaque fichier hash de 38 caractères est compressé on peut en lire le contenu par la commande

```
git cat-file -p hash
```

- L'option -p permet de déterminer le type du fichier automatiquement
- le hash ne doit pas contenir les 38 caractères mais commencer par les 2 caractères du répertoire suivi d'un nombre suffisant de caractères parmi les 38 le rendant unique

Méchanisme de versions

- La suite de commandes suivantes :

```
echo 'version 1' > test.txt
git hash-object -w test.txt
echo 'version 2' > test.txt
git hash-object -w test.txt
```

- provoque deux versions du même fichier c'est à dire deux fichiers sha1
- On comprend donc l'équivence entre les versions de chaque fichier commité et chaque sha
- La commande `git cat-file -t` montre le type de chaque fichier alors que l'option -p continue à en afficher le contenu

Les arbres

Signification d'un arbre

- Git crée un arbre à partir de l'état de la zone d'attente ou index et écrit une série d'objets arbre à chaque update
- La commande pour créer un arbre est la suivante :

```
git update-index --add --cacheinfo 100644  83baae6 test.txt
```

- Le mode 100644 correspond à un ficher normal,100755 à un exécutable, 120000 un lien symbolique, ce sont les seuls modes pour les fichiers
- La commande `git write-tree` écrit la zone d'attente dans un objet arbre

Présentation des arbres

- git stocke en sus des données, des arbres qui regroupent un répertoire pointant sur les fichiers stockés
- Un arbre peut contenir des fichiers ou des sous arbres
- Un arbre stocke également les droits d'accès des fichiers ou de sous arbres sous la forme de modes
- Pour afficher l'arbre correspondant au dernier commit de la branche master se note

```
master^{tree}
```

- On peut donc afficher cet arbre via la commande (après un git commit)

```
git cat-file -p master^{tree}
```

- Si un SHA1 correspond à un arbre la commande suivante renvoie ''tree"

```
git cat-file -t 29615b9fedf2ab0c7449cb945b3ebd4d01b3b18b
```

Visualisation d'un arbre

- La commande suivante

```
git cat-file -p 29615b9fe
```

- renvoie un affichage en colonne indiquant le mode, le type, le SHA1 et le nom du fichier

```
040000 tree de4228c90071e47cf99ea0797b11bafac329e856    Common
100644 blob 6b39d0ea8b79fa33f68eb2bbf1a56d7b8b6c9845    Makefile
040000 tree 310532a2010ae9b60d30b2d55a0b84b224bf601e    anagramme
040000 tree 8ada075634775519ebf34f5708b7e8c2268b4dbd    essai
040000 tree 1f5a791db0c28ad9222d1f49dd0c755450d6df51    gmsdbcompare
040000 tree 30304edd501ad356c91901eb6f79b1586255a54d    gmsedit
```

Arbres et commit

- Pour créer un commit des tous les arbres ajoutés précédemment on exécute la commande :

```
echo 'first commit' |  git commit-tree d8329f
```

- Cette commande regroupe les différents ajouts et les rassemble sous un seule SHA1
- On peut visualiser le commit par la commande

```
git cat-file -p fdf4fc3
```

- qui affichera alors une information de la forme

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
```

Références

Références

- Le répertoire .git/refs contient deux répertoires : heads et tags
- heads contient les références, on peut créer une référence par la commande suivante

```
echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

- Ainsi les deux commandes suivantes deviennent équivalentes

```
git log --pretty=oneline  master
git log --pretty=oneline  1a410efb
```

- Git propose une manière sûre de mettre à jour une référence, c'est la commande update-ref

```
git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

La branche HEAD

- Le fichier .git/HEAD est une référence symbolique à la branche courante
- Son contenu n'est pas une empreinte SHA1 mais la ligne suivante

```
ref: refs/heads/test
```

- Quand on exécute un git commit, l'objet commit est créé en spécifiant le parent de cet objet commit quelle que soit l'empreinte SHA-1 pointée par la référence de HEAD.
- On peut éditer manuellement ce fichier, mais il existe une commande plus sûre pour le faire : symbolic-ref. Vous pouvez lire le contenu de votre fichier HEAD avec cette commande :

```
git symbolic-ref HEAD
refs/heads/master
```

- On peut également définir la valeur du HEAD par

```
git symbolic-ref HEAD refs/heads/test
```

Opérations courantes

Supprimer un fichier de l'historique

- Pour lister les fichiers (ici on choisit le fichier core dont on a remarqué la présence)

```
git rev-list --all --objects | grep core
9eb2eb4c217104ffca293660cce75eb8bce62ce9 Example/core
fd0c96e40d6bed37cd444a1f97c800a44ad68d26 bin/core
ad97f699ee59ff1fb2c87a8bc352b53f5dcaab07 Example/core
f452057be6ecf99216aa0f8ae5223b9e0f66eeef Example/core
```

- on va supprimer Example/core et bin/core

```
git filter-branch --force --index-filter 'git rm --cached --ignore-unmatch \
 Example/core' --prune-empty --tag-name-filter cat -- --all
git pull
git push
```

Chercher les gros fichiers

- La commande suivant trie suivant des données numériques (-n) sur la troisième clé (-k) et donne les trois fichiers les plus volumineux (tail -3)

```
git verify-pack -v .git/objects/pack/pack-29...69.idx  | sort -k 3 -n  \
     | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

- Pour voir le nom du fichier correspondant au blob le plus gros c'est à dire le 82c99a3

```
git rev-list --objects --all | grep 82c99a3
```

Annexe

Rubriques

- Gitolite
- Références

Gitolite

Présentation gitolite

- Gitolite est un utilitaire permettant de gérer les dépôts et les utilisateurs sur une machine. Par défaut il ne gère pas les accès publics mais les accès par nom d'utilisateur ou par groupe
- On l'installe sur un serveur en tant qu'utilisateur git via la commande

```
git clone http://github.com/sitaramc/gitolite.git
```

- Une fois installé suivre la documentation d'installation qui indique de faire :

```
mkdir -p $HOME/bin
./gitolite/install -to $HOME/bin
$HOME/bin/gitolite setup -pk /home/votrenom/.ssh/id-rsa.pub # la clé publique e
git clone votrenom@locahost:gitolite-admin # où host est l'adresse ip ou le nom
```

- en général on crée un compte git pour ne pas que les accès à gitolite se fasse par votrenom mais par git

Créer un dépôt avec gitolite

- Dans le répertoire gitolite-admin on trouve deux répertoire :
  - le répertoire **keydir** des clés publiques authorisées à se connecter
  - le répertoire **conf** de configuration des différents dépôts contenant un fichier gitolite.conf
- Pour ajouter un utilisateur mettre sa clé sous la forme prenom.key ou nom.key dans le répertoire **keydir**
- Ne pas oublier de faire un git add sur ce fichier
- Dans le fichier gitolite.conf du réperoire **conf** ajouter l'utilisateur en face des droits RW+ (+ pour rewind permissions) RW ou R (séparés par des blancs)
- Pour ajouter un projet ajouter les lignes

```
repo nomdudepot
     RW+ = admin
```

- Commiter et pousser la modification

Groupes avec gitolite

- Pour ajouter un groupe, il suffit de le définir dans le fichier gitolite.conf en l'ajoutant au début du fichier comme ceci : @nomgroupe = user1 user2 user3
- Puis au lieu de la liste des utilisateurs on entre @nomgroupe
- Ne pas oublier de commiter et de pousser la modification

Accéder au dépôt

- Pour accéder au dépôt pour un utilisateur enregistré :

```
git clone utilisateur@adresseserveur:nomdudepot
```

Gérer les dépots et les utilisateurs avec gitolite

- Pour ajouter un projet :
  - depuis votre compte ajouter le projet dans le fichier gitolite-admin/conf/gitolite.conf
  - git commit -am “ajout projet”
  - git push
- Pour ajouter un utilisateur :
  - copier sa clé dans gitolite-admin/keydir/utilisateur.pub
  - git add gitolite-admin/keydir/.
  - dans le fichier gitolite-admin/conf/gitolite.conf l’utilisateur apparait sous le nom de fichier sans l’extension pub
  - git push
- Chaque utilisateur pourra accéder au projet auxquel il est inscrit par la commande

```
git clone git@host:projet.git
```

Références

Quelques ressources en ligne

- Pro git : documentation pédagogique très conceptuelle
  http://git-scm.com/book/fr/v2
- La référence de pro git : donne totes les options en anglais
  https://git-scm.com/docs
- Git Immersion : 53 fiches appelées LAB http://gitimmersion.fr/
- Git Magic :
  http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/fr/
- Référence visuelle de Git :
  http://marklodato.github.io/visual-git-guide/index-fr.html

Livres en français

- Pro Git (version papier ou epub idenique à la ressource en ligne ou PDF)
- Git Pocket Guide9 - août 2013 de Richard Silverman chee ORelly
- Mémento Git à 100% - avril 2012 de Raphaël Hertzog et Pierre Habouzit