

Gestion des sources avec GIT

David THIBAU – 2020

david.thibau@gmail.com



Agenda

- Introduction

- SCMs
- Le projet Git et ses concepts
- Installation et démarrage

- Les bases de Git

- Création d'un dépôt
- Enregistrer des modifications
- Visualiser l'historique
- Annuler des actions

- Git et les branches

- Les branches Git
- Brancher et fusionner
- Visualiser les branches
- Rebaser
- Typologies de branches
- Les tags

- Le serveur Gitlab

- Introduction
- Projets et Membres
- Les issues

- Workflows de collaboration

- Les dépôts distants
- Les branches distantes
- Patterns de collaboration
- GitlabFlow et Merge Request

- Pour aller plus loin

- Les refs
- Quelques outils utiles
- Hooks et personnalisation

- Annexes

- Les sous-modules
- Migration depuis SVN



Introduction

Les SCMs
Le projet Git et ses concepts
Installation et démarrage



SCM

Un **SCM** (*Source Control Management*) est un système qui enregistre les changements faits sur un fichier ou une structure de fichiers afin de pouvoir revenir à une version antérieure

Le système permet :

- De restaurer des fichiers
- Restaurer l'ensemble d'un projet
- Visualiser tous les changements effectués et leurs auteurs



Types de fichiers

La plupart du temps les SCMs sont utilisés pour les fichiers sources des développeurs bien qu'ils soient capable de traiter **tout type** de fichiers

- Par exemple, un web designer peut vouloir garder toutes les versions d'une image ou d'une maquette de page
- Cependant, les SCMs sont associés à des outils de comparaison de version. Ces outils fonctionnent correctement avec les formats textes



SCM Local

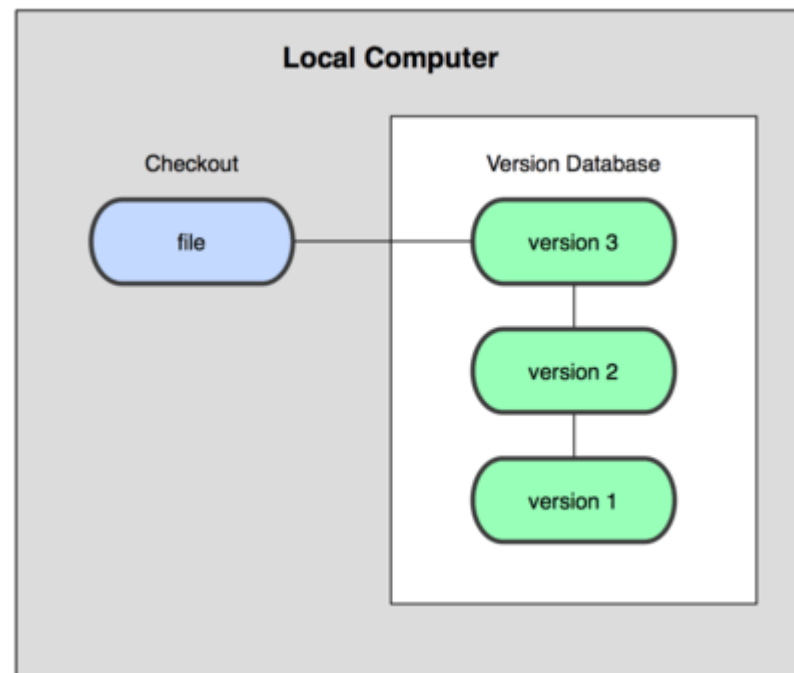
Les SCMs **locaux** ont une simple base de données qui garde tous les changements effectués sur les fichiers.

- Ces outils stockent, généralement, des ***patches*** (différences entre 2 versions) dans un format spécifique. Il peut ainsi recréer un fichier dans une révision particulière

RCS (inclut par exemple dans MacOSX) est un SCM local.

SCCS dans le monde Unix

SCM Local





SCM centralisé

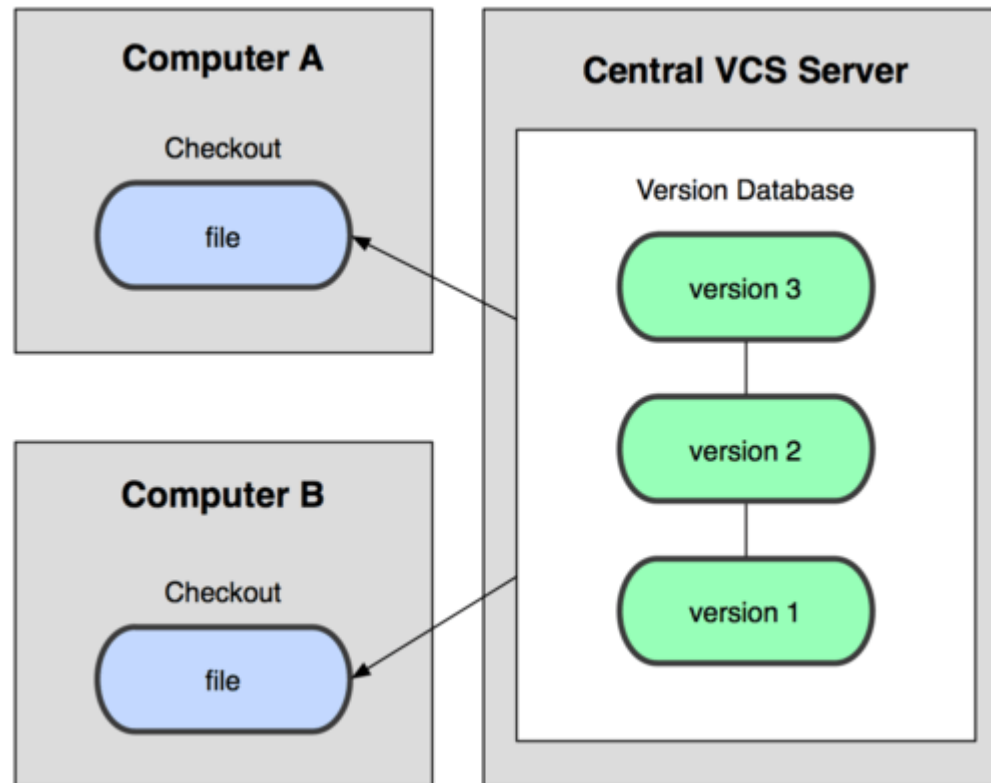
Les SCMs **centralisés** permet aux développeurs de collaborer sur les mêmes fichiers.

- Un serveur unique contient tous les fichiers revisionnés et des clients s'y connectent pour récupérer les fichiers et enregistrer des changements
- Exemples : CVS, Subversion et Perforce

Les avantages de ce type de solution :

- Chacun est au courant de ce que font les autres
- Les administrateurs contrôlent finement et facilement les permissions de chaque client

SCM centralisé





Inconvénients

L'inconvénient le plus évident est la dépendance de tous les clients envers le serveur.

- Si le serveur est défaillant, personne ne peut collaborer, ni sauvegarder ses changements
- De nombreuses opérations courantes nécessitent le réseau et sont relativement lentes (Accès à l'historique, comparaison de version, commit, ...)

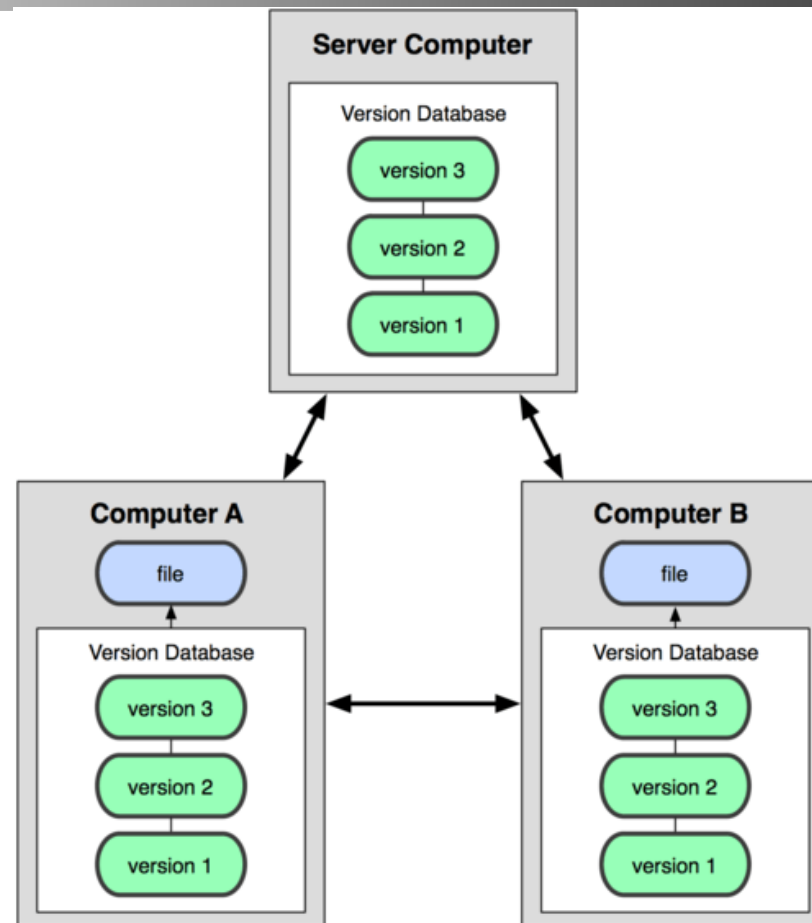


SCM distribué

Avec les SCMs **distribués**, les clients ne récupèrent pas le dernier instantané des fichiers mais l'intégralité du dépôt ou référentiel

- Ainsi si un serveur défaille, les clients peuvent recréer le référentiel à partir de leur copie locale et continuer à collaborer
- Le fait de disposer de plusieurs référentiels distants permet de collaborer avec différents groupes de personnes de façon différente et de mettre en place différents workflows

SCM distribué





Le projet Git



Historique

Le projet Open Source Linux impliquant de nombreux développeurs est à l'origine de *Git* :

- *De 1991-2002* : Les changements étaient gérés via des patches et des fichiers archivés
- *2002* : le projet commence à utiliser un système propriétaire distribué gratuitement *BitKeeper*
- *2005* : Les accords commerciaux avec *BitKeeper* changent et la communauté Linux (en particulier Linus Torvalds) est poussée à développer leur propre outil en bénéficiant des leçons apprises avec *BitKeeper* : ***Git***



Objectifs de Git

Les objectifs initiaux de *Git* sont alors :

- La vitesse
- Un design simple
- Un support efficace pour le développement non-linéaire (des milliers de branches parallèles)
- Entièrement distribué
- Capable de gérer efficacement de gros projets comme Linux (vitesse et volume de données)

=> Ces objectifs ont été conservés lors des différentes versions de Git



Différence d'approche

Git adopte une approche radicalement différente pour le stockage des données par rapport aux systèmes traditionnels comme Subversion

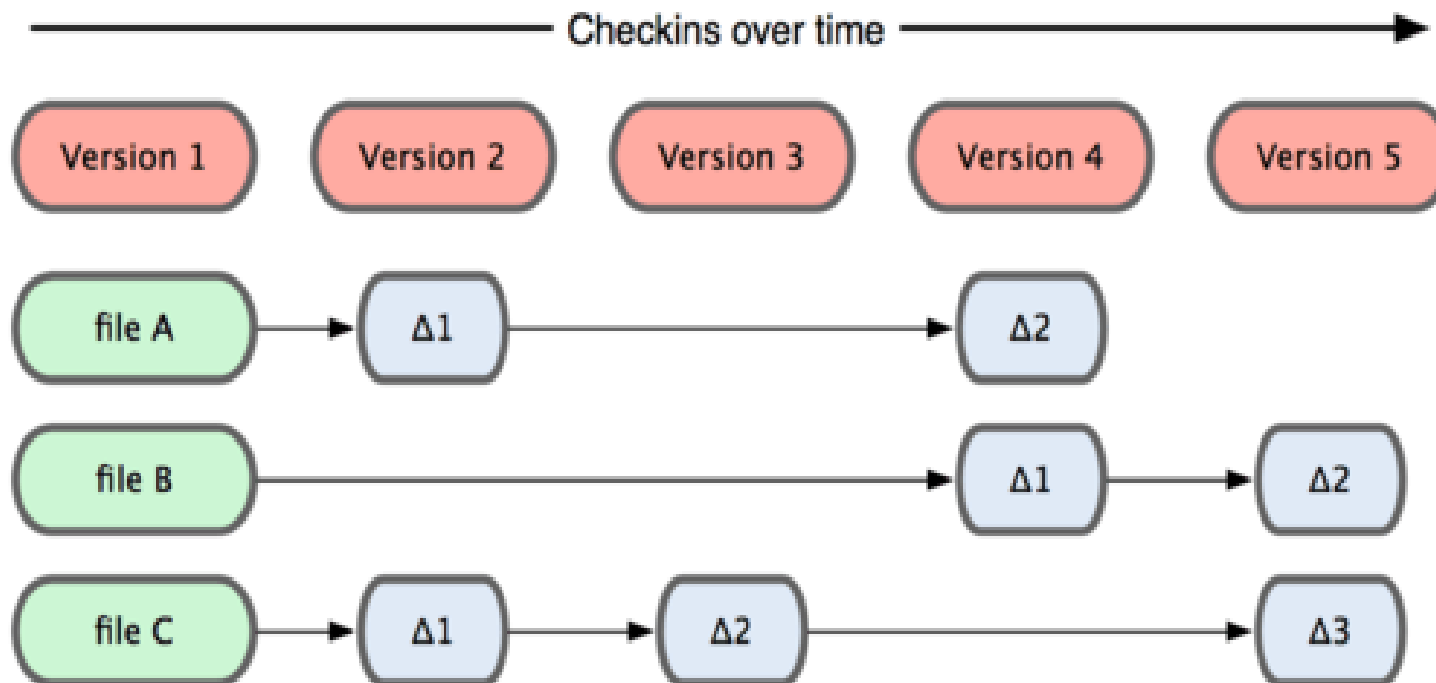
Au lieu de stocker les fichiers initiaux et les changements entre révisions, *Git* stocke des **instantanés complets**

- A chaque commit, *Git* prend un instantané de l'état des fichiers et le stocke dans sa base.
- Pour être efficace, si un fichier est inchangé, son contenu n'est pas stocké une nouvelle fois mais plutôt une référence au contenu précédent

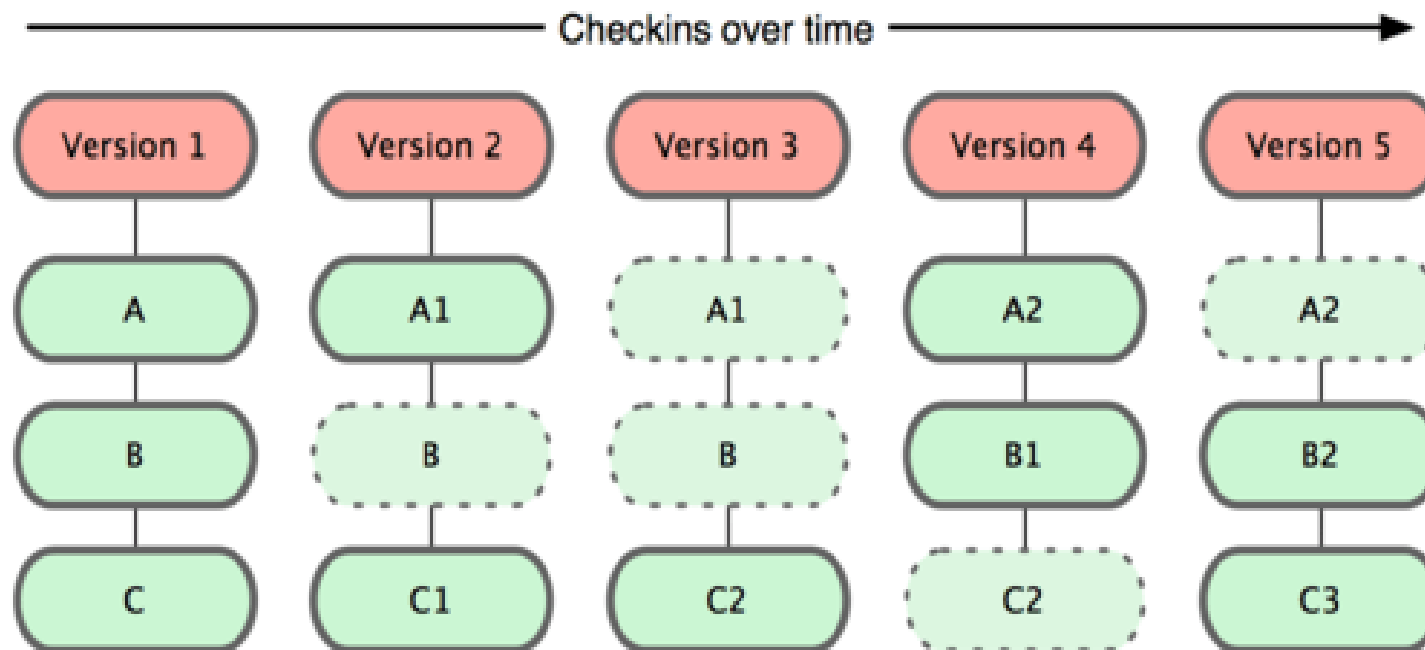
=> *Cette approche fait que Git se comporte plutôt comme un mini système de fichiers proposant des outils très efficaces*



Approche standard



Approche Git





Opérations locales

La plupart des opérations Git nécessitent seulement des fichiers **locaux** ; aucune information provenant d'un ordinateur distant n'est nécessaire.

La plupart des opérations sont donc instantanées.

- Par exemple, parcourir l'historique d'un fichier, calculer les différences entre 2 versions, committer, etc ...

Cela signifie également que l'on peut facilement travailler *offline*



Intégrité

Toutes les données du référentiel *Git* sont associées à un **checksum** avant qu'elles soient stockées. Le check-sum constitue l'identifiant de la donnée Git.

- Le *checksum* est un *hash* SHA-1 constitué de 40 caractères hexadécimaux fonction du contenu d'un fichier ou d'un répertoire.

Exemple :

24b9da6552252987aa493b52f8696cd6d3b00373

Les fichiers sont donc stockés dans le référentiel *Git* non pas par leur noms mais par leur clés de hachage

- Il est ainsi impossible de changer le contenu d'un fichier sans que *Git* s'en aperçoive



Seulement des ajouts

La plupart des opérations dans Git consistent à ajouter des informations dans la base de données

- Ainsi, il est très difficile de faire des actions irréversibles

Comme tout SCM, il est possible de perdre des changements qui n'auraient pas été committés, par contre il est difficile de perdre des données surtout si on les pousse régulièrement vers un autre référentiel



État des fichiers

Les fichiers gérés par Git peuvent avoir 3 états :

- **Committed** : Les données sont stockées dans la base de données locale
- **Modified** : Le fichier a été changé mais pas encore committé dans la base
- **Staged** : Le fichier modifié a été marqué comme faisant partie du prochain commit



Sections d'un projet

Ces 3 statuts fait qu'un projet Git est décomposé en 3 sections :

- Le **répertoire Git (.git/)** est l'endroit où Git stocke les métadonnées et les objets de sa base de données. Il contient l'intégralité des informations
- Le **répertoire de travail** est un « checkout » d'une version du projet. Les fichiers sont extraits de la base de données compressée et peuvent ensuite être modifiés
- La **zone de staging** est un simple fichier (quelquefois nommé **index**) qui stocke les informations sur ce qu'il faut inclure dans le prochain commit.



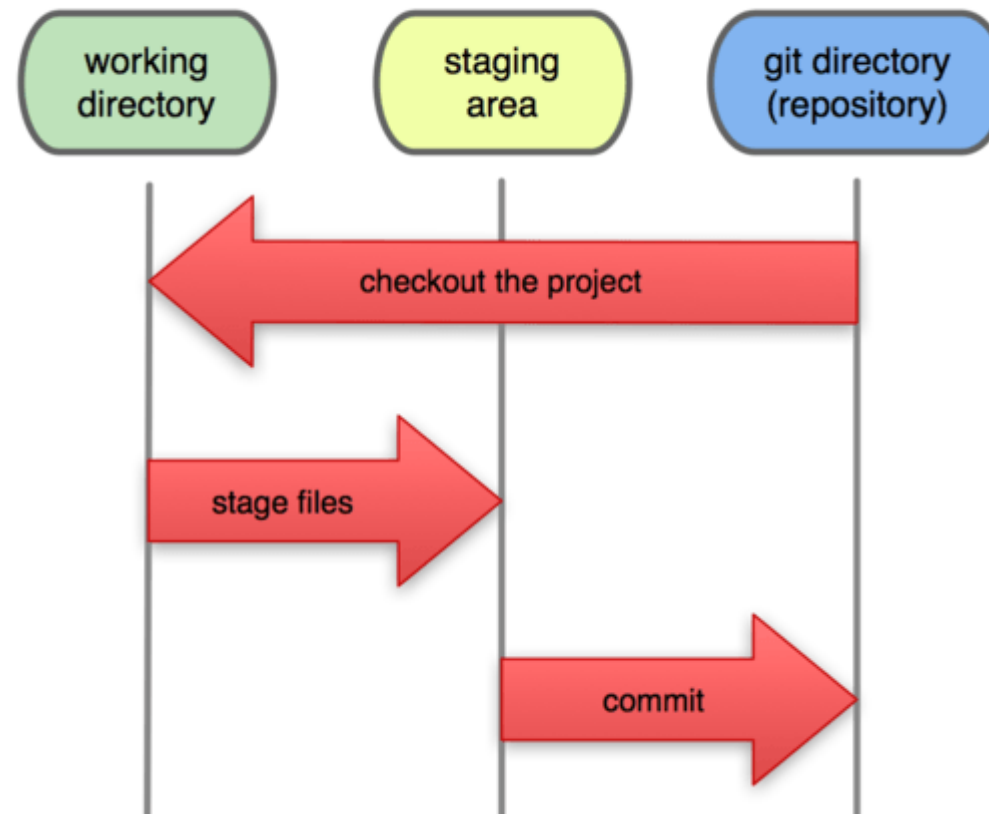
Workflow standard

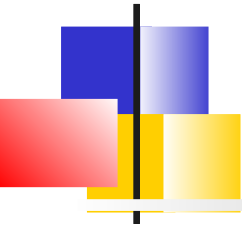
Le workflow standard de Git est :

1. Les fichiers du répertoire de travail sont **modifiés**
2. Ils sont ensuite placées dans la zone de **staging**.
3. Au **commit**, les fichiers de la zone de staging sont stockés dans le répertoire Git

Sections d'un projet Git

Local Operations





Installation



Installation d'un binaire

Linux

```
$ yum install git
```

```
$ apt-get install git
```

Mac

- Installeur en mode graphique :

- <http://sourceforge.net/projects/git-osx-installer/>

- Via MacPorts (+ = Extras)

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

Windows

Installateur du projet *msysGit*

- <http://msysgit.github.io>

L'installateur installe un outil de commande en ligne style Unix incluant un client SSH (plus simple à utiliser que les commandes natives Windows) et une interface graphique



Mise en place

Après l'installation, la première chose à faire est de configurer son environnement grâce à l'outil ***git config***

Cet outil stocke des variables de configuration dans 3 emplacements :

- ***/etc/gitconfig*** : Contient les valeurs pour chaque utilisateur du système et leurs dépôts. L'option ***--system*** permet de lire et écrire vers ce fichier.
- ***~/.gitconfig*** : Valeurs spécifiques à un utilisateur. L'option ***--global*** permet les interactions avec ce fichier.
- ***.git/config*** dans le répertoire Git : Valeurs spécifiques à un dépôt.

Chaque valeur spécifiée surcharge la valeur spécifiée au niveau supérieur



Configuration Windows

Dans les systèmes Windows, Git cherche le fichier *.gitconfig* dans le répertoire *\$HOME*

Variable d'environnement *%USERPROFILE%*,

Typiquement *C:\Documents and Settings\%USER%*

Il cherche également */etc/gitconfig*
relativement au disque précisé lors de
l'installation de Git.



Variables à positionner

Les premières variables à positionner sont le nom de l'utilisateur et l'adresse email. Ces informations seront utilisées et visibles des collaborateurs lors des synchronisations de référentiels

```
$ git config --global user.name "John Doe"
```

```
$ git config --global user.email johndoe@example.com
```

L'éditeur texte par défaut

```
$ git config --global core.editor emacs
```

L'outil de différence (pour résoudre les conflits de merge) :

```
$ git config --global merge.tool vimdiff
```

Git supporte les outils suivants : *kdif3*, *tkdiff*, *meld*, *xxdiff*, *emerge*, *vimdiff*, *gvimdiff*, *ecmerge*, et *opendiff*

Pour voir toutes les options de configuration disponibles :

```
$ git config --help
```



Vérification de la configuration

git config --list permet de visualiser toutes les variables d'environnement de Git

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
```

...

- Il est possible de voir plusieurs fois la même clé, (si elle est définie à plusieurs niveau), seule la dernière valeur est utilisée par Git

Il est possible de visualiser les valeurs d'une clé par la commande
git config {key}



Aide

```
$ git help <verb>
```

```
$ git <verb> --help
```

```
$ man git-<verb>
```

Exemple : ***\$ git help config***

Il est possible également d'obtenir du support via le serveur IRC (*irc.freenode.net*) et les canaux #git ou #github



Les bases de Git

Création d'un dépôt
Enregistrer les modifications
Visualiser l'historique
Annuler des actions
Les tags



Création d'un dépôt



Créer un dépôt

Il y a 2 façons de créer un dépôt :

- **Importer** un projet existant dans Git
- **Cloner** un dépôt d'un autre serveur



Importer un projet existant

L'initialisation consiste à créer un sous-répertoire *.git* dans le projet contenant tous les fichiers nécessaires au dépôt:

```
$ git init
```

A ce moment, le dépôt est initialisé mais encore aucun des fichiers n'est suivi par Git

Pour ce faire, il faut ajouter les fichiers désirés et les committer.

Exemple :

```
$ git add *.c
```

```
$ git add README
```

```
$ git commit -m 'initial project version'
```



Cloner un dépôt

La commande ***git clone [url]*** permet de cloner un dépôt

L'ensemble des fichiers est alors répliqué et la copie peut servir de réplique à d'autres clients

Exemple :

```
$ git clone git://github.com/schacon/grit.git
```

=> Crée un répertoire *grit*

=> Crée le sous-répertoire *.git* initialisant le repository

=> Check-out un répertoire de travail prêt à être utilisé

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Idem mais le répertoire projet s'appelle *mygrit*

D'autres protocoles sont disponibles (*http(s)* , *ssh*)



Enregistrer les modifications



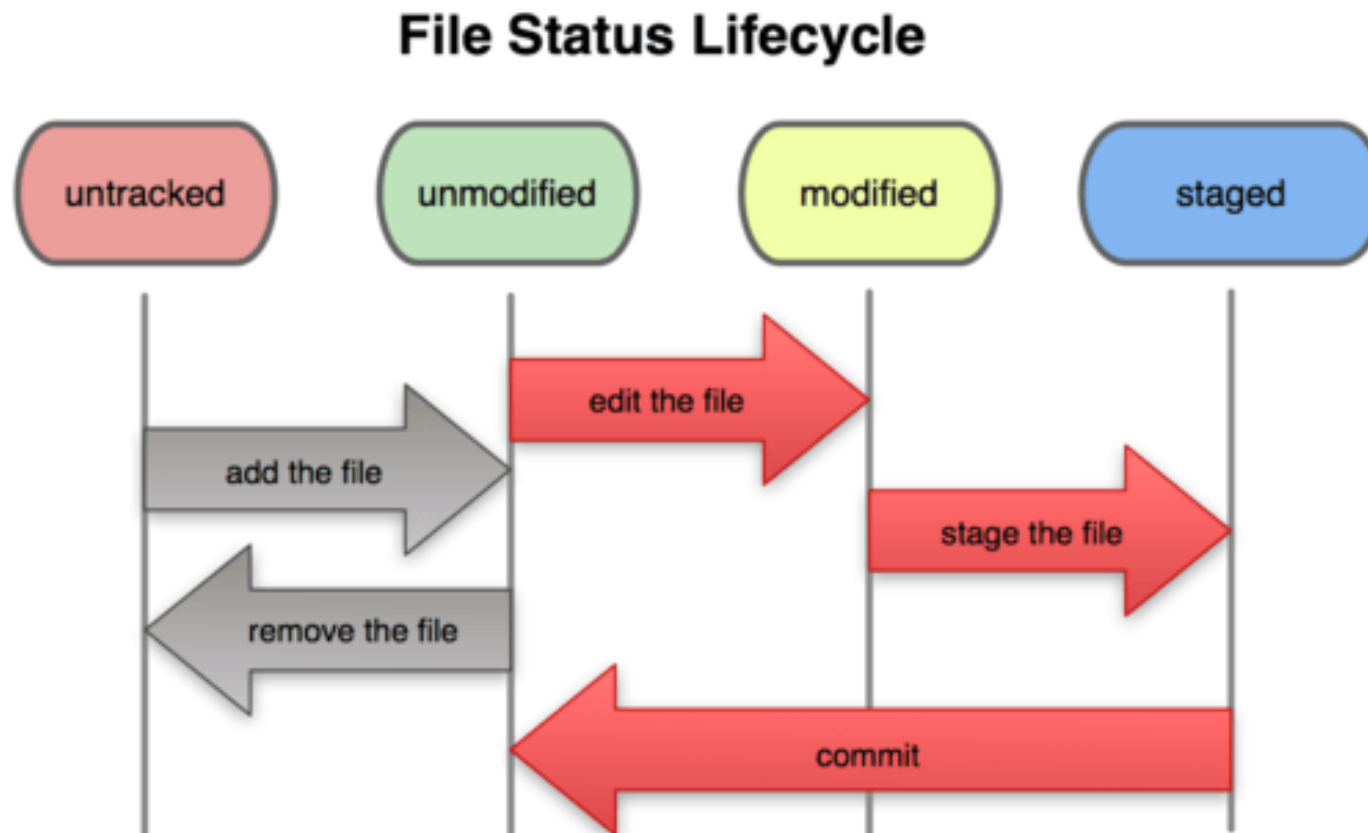
Fichiers du répertoire de travail

Chaque fichier du répertoire de travail peut être ***suivi*** ou ***non-suivi***.

Les fichiers suivis sont les fichiers présents dans le dernier instantané ; ils peuvent être dans les 3 statuts : ***non-modifié***, ***modifié*** ou ***indexé***

- Lors de l'édition d'un fichier, Git le détecte comme modifié
- Il faut alors passer ces fichiers dans la zone de staging ou index puis les committer

Statuts des fichiers





git status

L'outil principal pour vérifier le statut des fichiers est ***git status***

Par exemple, le résultat de cette commande après un clone :

```
$ git status
```

```
On branch master
```

```
nothing to commit, working directory clean
```

=> Aucun fichier suivi n'a été modifié sur la branche par défaut *master*



git status

Si par exemple, un fichier a été ajouté, le résultat est :

```
$ vim README
```

```
$ git status
```

```
On branch master
```

Untracked files:

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

README

nothing added to commit but untracked files present (use "git add" to track)



Ajouter des fichiers

Pour commencer à suivre un fichier, il faut donc utiliser la commande ***git add***.

Par exemple :

```
$ git add README
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

```
new file:   README
```

La commande *git add* prend en argument un chemin de fichier ou de répertoire.

Si il s'agit d'un répertoire, la commande ajoute tous les fichiers du répertoire récursivement



Indexation des fichiers modifiés

Après l'édition d'un fichier déjà suivi

```
$ vim benchmarks.rb
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README

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: benchmarks.rb



Indexation

Pour le passer en staged, il faut également exécuter la commande *git add* .

```
$ git add benchmarks.rb
```

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
new file:   README
```

```
modified:   benchmarks.rb
```

```
Both files are staged and will go into your next commit.
```



Modification d'un fichier indexé

Si on modifie un fichier indexé avant de le commiter

```
$ vim benchmarks.rb
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

```
new file:   README
```

```
modified:   benchmarks.rb
```

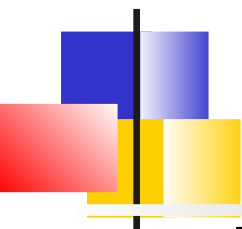
Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

```
modified:   benchmarks.rb
```

benchmarks.rb est alors listé comme staged **ET** unstaged. Si on effectue un commit, c'est la version staged qui sera incluse dans le dépôt (la dernière exécution de git add)



Visualiser les changements

La commande ***git diff*** permet de connaître exactement les modifications qui ont été faites

- Cette commande compare le contenu du répertoire de travail et le contenu de l'index. Le résultat montre les modifications effectuées qui n'ont pas encore été mises dans l'index.
- Par défaut, *git diff* ne montre que les changements qui ne sont pas indexés

Pour voir, ce qui est indexé et ce qui fera partie du prochain commit l'option ***--cached*** peut être utilisée



Exemple : *git diff*

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb // Header
index 3cb747f..da65585 100644 // <plage SHA> <mode : type et permissions>
--- a/benchmarks.rb // Fichier d'origine
+++ b/benchmarks.rb // Nouveau fichier
@@ -36,6 +36,10 @@ def main // Une partie ou il y a des différence
    @commit.parents[0].parents[0].parents[0]
  end

+   run_code(x, 'commits 1') do // ligne ajoutée
+     git.commits.size
+   end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```



Exemple : *git diff --cached* (ou *--staged*)

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 00000000..03902a1
--- /dev/null // Le fichier n'existait pas
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git
  repository
```



Commit

La commande *commit* n'a d'effet que sur l'index.

Tous les fichiers créés ou modifiés qui n'ont pas été ajoutés ne participent pas au commit

git commit démarre l'éditeur spécifié par la variable d'environnement *\$EDITOR* qui a pu être configuré par la commande *git config --global core.editor*

L'éditeur contient un message par défaut (la sortie de la commande *git status* en commentaire) et une ligne vide.

On peut modifier le contenu du fichier temporaire pour adapter le message



Example (*vi*)

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the
# commit.
#
# On branch master
#
# Changes to be committed:
#       new file:   README
#       modified:   benchmarks.rb
#
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```



Options

L'option **-v** permet de positionner les différences dues aux changements dans l'éditeur

Il est possible de passer directement le message en ligne avec l'option **-m**

```
$ git commit -m "Story 182: Fix benchmarks for speed"
```

Avec l'option **-a** la commande ajoute automatiquement les fichiers déjà suivis dans le commit (*git add* n'est alors plus nécessaire)



Résultat du commit

La commande *commit* affiche des informations :

- La branche que l'on a committée
- Le checksum du commit
- Combien de fichiers ont été committés
- Des statistiques sur les lignes ajoutés et supprimés dans le commit

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 3 insertions(+)
 create mode 100644 README
```



Suppression de fichiers

Pour supprimer un fichier de Git, il faut l'enlever des fichiers suivi puis committer

La commande ***git rm*** est alors utilisée ; elle a l'avantage de supprimer également les fichiers du répertoire de travail



Cas de suppression

Si l'on supprime un fichier du répertoire de travail, le changement est détecté par Git mais il n'est pas présent dans l'index

- Le résultat de *git status* l'indique dans la zone “Changes not staged for commit”

Ensuite si on exécute *git rm*, la suppression du fichier est mise dans l'index

- => au prochain commit, le fichier sera supprimé du dépôt

Si le fichier a été modifié auparavant et mis dans l'index, il faut forcer la suppression avec l'option **-f**.



Conserver un fichier

L'option **--cached** permet de conserver un fichier sur son disque en enlevant le suivi par Git.

```
$ git rm --cached readme.txt
```

Cela peut être utile si on a oublié de l'inclure dans les fichiers à ignorer (*.gitignore*) et qu'il est été ajouté accidentellement dans *Git*



Suppression de plusieurs fichiers

Il est possible d'indiquer un répertoire ou un ensemble de fichiers à la commande *git rm*

- Par exemple, pour supprimer tous les fichiers qui ont l'extension *.log* dans le répertoire *log*

```
$ git rm log/\*.log
```

La notation backslash (\) est nécessaire dans un environnement non Windows

- Pour supprimer tous les fichiers se terminant par *~*

```
$ git rm \*~
```



Déplacements

A la différence des autres SCM, Git ne suit pas explicitement les déplacements de fichiers. Si l'on renomme un fichier, aucune métadonnée spécifique n'ait enregistrée

Cependant la commande de déplacement ***git mv*** permet à Git de détecter un renommage (via le checksum)

```
$ git mv README README.txt
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

```
renamed:    README -> README.txt
```

Cette commande est équivalente à un déplacement natif suivi d'une suppression et d'un ajout

```
$ mv README README.txt
```

```
$ git rm README
```

```
$ git add README.txt
```



Ignorer des fichiers

Les fichiers présents dans l'arborescence de travail que l'on ne veut pas que Git suive doivent être marqués comme à ignorer (les fichiers compilés, de trace, de configuration spécifique au poste de travail, ...)

Les motifs spécifiant les fichiers à ignorer (patterns) sont précisés dans le fichier ***.gitignore***

=> Il est préférable de mettre en place le fichier ***.gitignore*** au démarrage du projet



Règles de syntaxe

Les règles de syntaxe pour les motifs sont :

- Les lignes vides ou démarrant par le caractère # sont ignorées.
- Il est possible de terminer les motifs par / pour spécifier un répertoire
- Le point d'exclamation permet d'exprimer le contraire d'un motif
- Les motifs sont des expressions régulières simplifiées :
 - L'astérisque (*) représente zéro ou plusieurs caractères;
 - [abc] représente n'importe quel caractère spécifié entre crochets (dans ce cas a, b, ou c);
 - le caractère ? Représente un unique caractère
 - des crochets englobant 2 caractères séparés par un tiret ([0-9]) représentent n'importe quel caractère de l'intervalle (dans ce cas un chiffre de 0 à 9) .

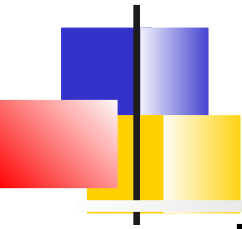


Exemple *.gitignore*

```
# a comment - this is ignored
# no .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the root TODO file, not subdir/TODO
/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```



Visualiser l'historique



Historique

La commande ***git log*** affiche les commits effectués dans le dépôt dans l'ordre chronologique inverse

La commande liste chaque commit avec :

- Son checksum SHA-1
- Le nom de l'auteur
- L'email
- La date du commit
- Le message du commit



Exemple

```
$ git log
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
changed the version number
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Sat Mar 15 16:40:33 2008 -0700
```

```
removed unnecessary test code
```

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Sat Mar 15 10:31:28 2008 -0700
```

```
first commit
```



Options

La commande log propose de nombreuses options :

- **-p** : Affiche les différences introduites par chaque commit
- **--word-diff** : Affiche les différences au niveau mot plutôt que ligne. Utile pour les documents textes plutôt que du code
- **--stat** : Affiche des statistiques pour chaque fichier modifié
- **--shortstat** : Affiche seulement les statistiques des changements de ligne de l'option --stat
- **--name-only** : Affiche le nom des fichiers modifiés après le commit
- **--name-status** : Affiche la liste des fichiers avec les informations added/modified/deleted
- **--abbrev-commit** : Affiche seulement les premiers caractères du checksum
- **--relative-date** : Affiche la date dans un format relatif (par exemple, “2 weeks ago”)
- **--graph** : Affiche un graphe ASCII de la branche et des historiques de fusion.
- **--pretty** : Permet de contrôler le format d'affichage des commits (oneline, short, full, fuller, et format (on spécifie alors le format voulu).
- **--oneline** : Un raccourci pour `--pretty=oneline --abbrev-commit`.



Exemple -p

```
$ git log -p -1
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
changed the version number
```

```
diff --git a/Rakefile b/Rakefile
```

```
index a874b73..8f94139 100644
```

```
--- a/Rakefile
```

```
+++ b/Rakefile
```

```
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
```

```
spec = Gem::Specification.new do |s|
```

```
  s.name      = "simplegit"
```

```
-  s.version  = "0.1.0"
```

```
+  s.version  = "0.1.1"
```

```
  s.author   = "Scott Chacon"
```

```
  s.email    = "schacon@gee-mail.com"
```



Example *--word-diff*

```
$ git log -U1 --word-diff
```

```
commit ca82a6dff817ec66f44342007202690a93763949
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date:   Mon Mar 17 21:52:11 2008 -0700
```

```
    changed the version number
```

```
diff --git a/Rakefile b/Rakefile
```

```
index a874b73..8f94139 100644
```

```
--- a/Rakefile
```

```
+++ b/Rakefile
```

```
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
```

```
    s.name      = "simplegit"
```

```
    s.version    = ["0.1.0"]{+"0.1.1"+}
```

```
    s.author     = "Scott Chacon"
```



Exemple *--pretty=format*

```
$ git log --pretty=format:"%h - %an, %ar : %s"
```

```
ca82a6d - Scott Chacon, 11 months ago : changed the  
version number
```

```
085bb3b - Scott Chacon, 11 months ago : removed  
unnecessary test code
```

```
a11bef0 - Scott Chacon, 11 months ago : first commit
```



Option pour le formattage

- **%H** Commit hash
- **%h** Commit hash abrégé
- **%T** Hash de l'arborescence de répertoire
- **%t** Hash de l'arborescence de répertoire abrégé
- **%P** Hashes des branches parentes
- **%p** Hashes des branches parentes abrégés
- **%an** Nom de l'auteur
- **%ae** Email de l'auteur
- **%ad** Date de modification
- **%ar** Date de modification au format relatif
- **%cn** Nom du committeur
- **%ce** Email du committeur
- **%cd** Date du commit
- **%cr** Date du commit relative
- **%s** Sujet / Message



Options de limitation

Les options de limitation permettent de limiter le nombre de commits affichés :

- **-(n)** Les derniers n commits
- **--since, --after** : Limiter les commits après une date
- **--until, --before** : Limiter les commits avant une date.
- **--author** : Limiter les commits à un auteur
- **--committer** : Limiter les commits à un committer



Exemple

```
$ git log --after="2013-04-29T17:07:22+0200" --before="2013-04-29T17:07:22+0200" \
--pretty=fuller
```

```
commit de7c201a10857e5d424dbd8db880a6f24ba250f9
Author:      Ramkumar Ramachandra <artagnon@gmail.com>
AuthorDate:  Mon Apr 29 18:19:37 2013 +0530
Commit:      Junio C Hamano <gitster@pobox.com>
CommitDate:  Mon Apr 29 08:07:22 2013 -0700
```

git-completion.bash: lexical sorting for diff.statGraphWidth

df44483a (diff --stat: add config option to limit graph width, 2012-03-01) added the option diff.startGraphWidth to the list of configuration variables in git-completion.bash, but failed to notice that the list is sorted alphabetically. Move it to its rightful place in the list.

```
Signed-off-by: Ramkumar Ramachandra <artagnon@gmail.com>
Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

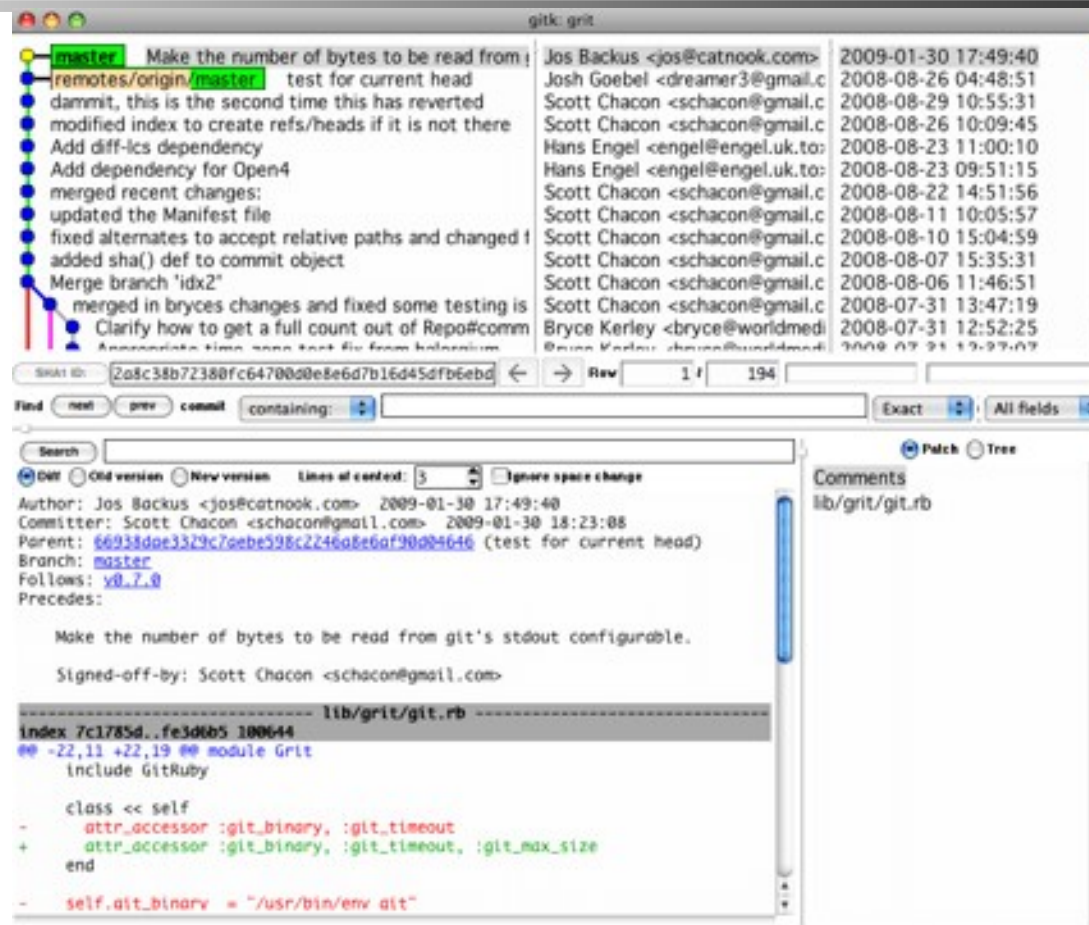



Exemple

```
$ git log --pretty="%h - %s" --author=gitster \  
    --after="2008-10-01T00:00:00-0400" \  
    --before="2008-10-31T23:59:59-0400" --no-merges  
-- t/
```

```
5610e3b - Fix testcase failure when extended attribute  
acd3b9e - Enhance hold_lock_file_for_{update,append}()  
f563754 - demonstrate breakage of detached checkout wi  
d1a43f2 - reset --hard/read-tree --reset -u: remove un  
51a94af - Fix "checkout --track -b newbranch" on detac  
b0ad11e - pull: allow "git pull origin $something:$cur
```

gitk





Annuler des actions



Changer le dernier commit

Si le commit a été effectué trop vite et que certains fichiers ont été oubliés ou que le message associé n'était pas approprié, il est toujours possible de le modifier grâce à l'option **--amend**

```
$ git commit --amend
```

=> Si aucun changement n'a été effectué, alors il sera possible de modifier le message

Pour ajouter des fichiers oubliés :

```
$ git commit -m 'initial commit'
```

```
$ git add forgotten_file
```

```
$ git commit --amend
```

Après ces commandes, il n'y a qu'un seul commit dans la base Git



Enlever un fichier de l'index

Pour enlever un fichier de l'index, il est possible d'utiliser la commande ***git reset***

```
$ git reset HEAD benchmarks.rb
```

```
Unstaged changes after reset:
```

```
M      benchmarks.rb
```

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
modified:   README.txt
```

```
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:   benchmarks.rb
```



Annuler les modifications d'un fichier

Annuler des modifications afin de récupérer la version du dernier commit s'effectue avec la commande **git checkout** (indiquée dans la sortie de *git status*)

```
$ git checkout -- benchmarks.rb
```

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
    modified:   README.txt
```

Attention, c'est une commande dangereuse, les modifications sur le fichier sont définitivement perdues



Git et les branches

Les branches Git
Brancher et fusionner
Visualiser les branches
Rebaser
Typologie de branches
Tags



Les branches Git



Introduction

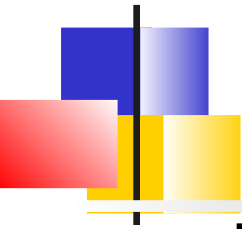
Le branchement signifie que le code diverge de la ligne principale de développement et que les deux branches évoluent indépendamment

Dans les autres outils de SCM, l'opération de branchement est généralement lourde car elle nécessite la création d'une nouvelle copie des sources

Les branches Git sont par contre très légères et les opérations de création et de basculement instantanées

=> Git encourage donc des workflows avec des branchements et des fusions de branches nombreuses (plusieurs fois dans la même journée).

=> Bien maîtriser cette fonctionnalité de Git peut avoir des impacts sur la façon de développer et de collaborer entre développeurs

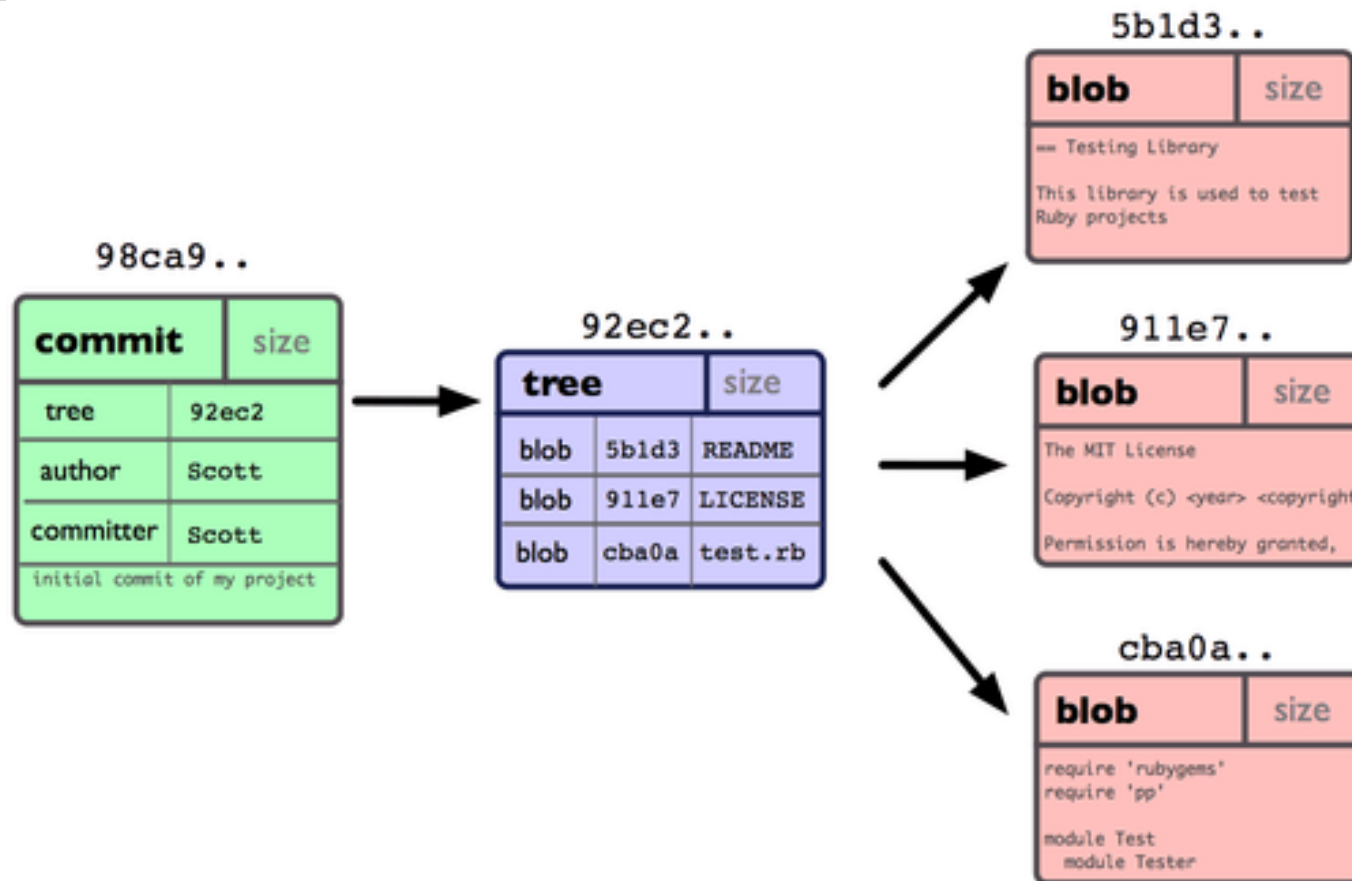


Rappels

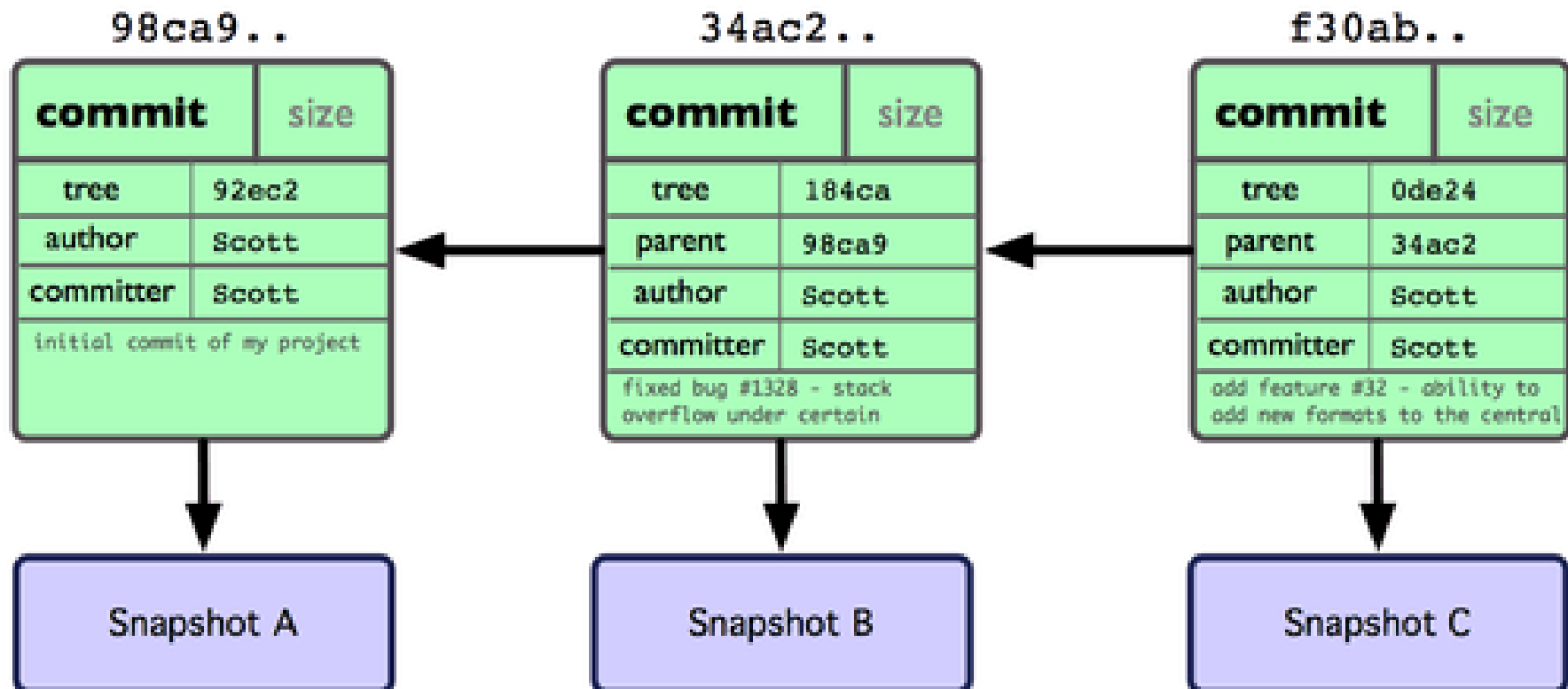
Dans la base Git, un objet *commit* est :

- Un **pointeur** vers l'instantané du contenu
- Les **méta-données** : auteur, message, ...
- 0 ou **plusieurs pointeurs vers les *commits* parents** :
 - 0 pour le premier commit
 - 1 pour un commit standard
 - Plusieurs pour un commit provenant d'une fusion de plusieurs branches

Exemple 3 fichiers committés



Commit successors





Création de branche

Une branche Git est simplement un **pointeur** pouvant se déplacer sur les commits du référentiel.

- La branche par défaut est nommé *master*.

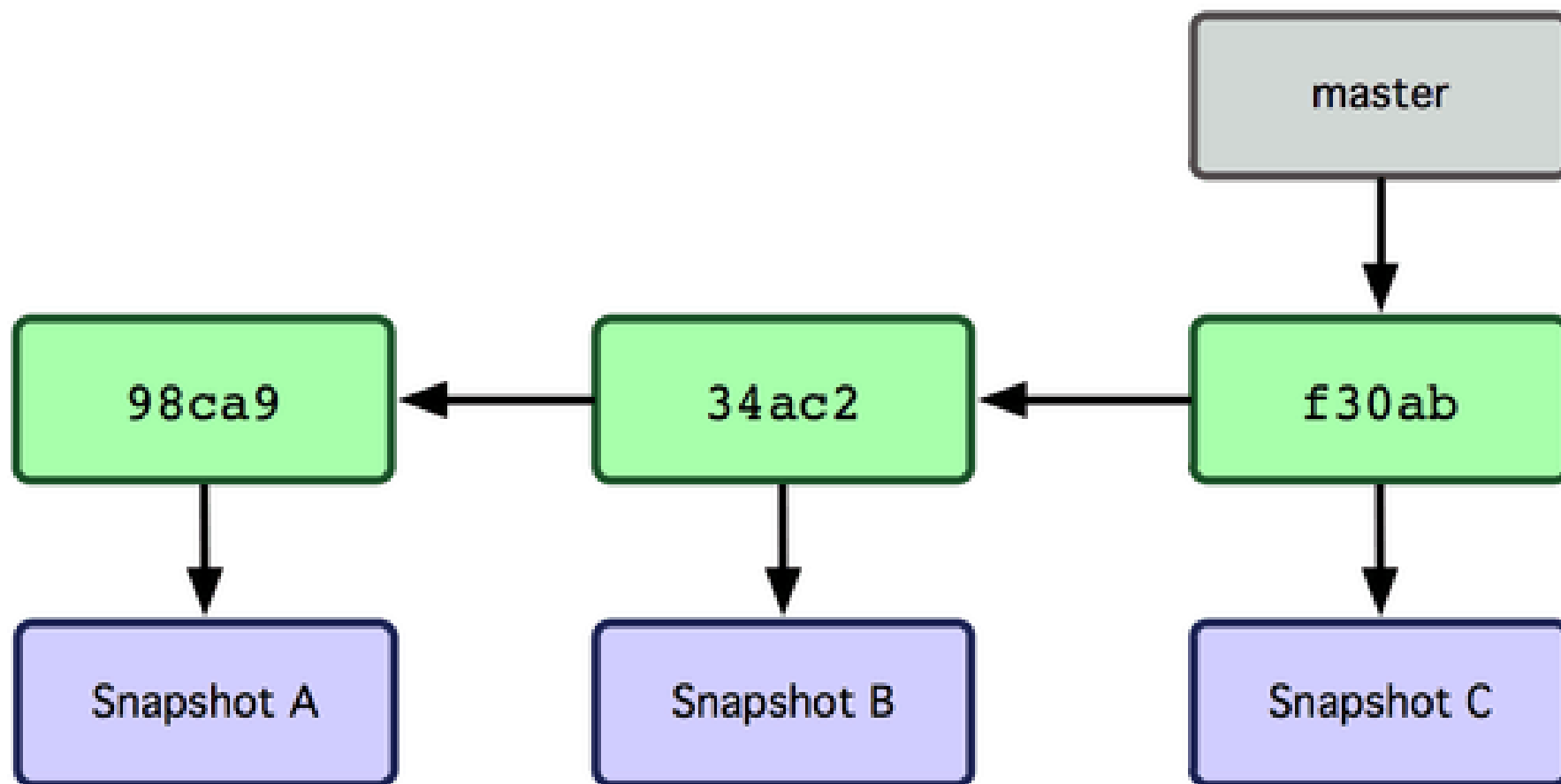
Lors des commits, ce pointeur se déplace vers le dernier commit

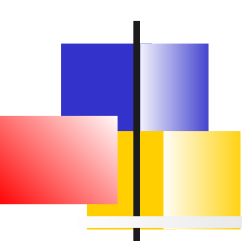
A la création d'une nouvelle branche, Git crée un nouveau pointeur portant le nom de la branche qui pointe sur le même dernier commit

```
$ git branch testing
```

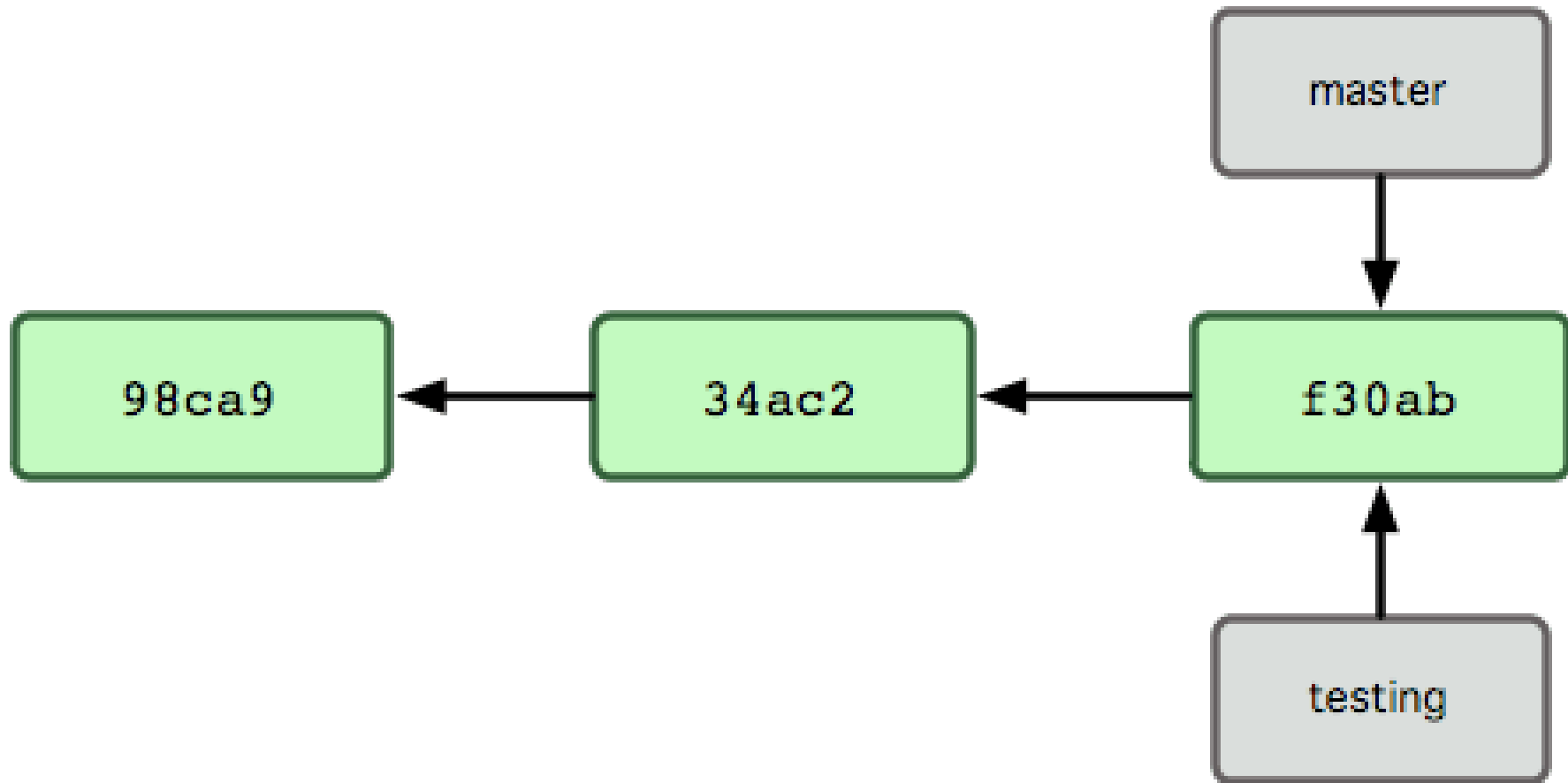


Branche master par défaut





Après création d'une branche





Avantages des pointeurs

Une branche Git est en fait un simple fichier contenant les 40 caractères du checksum SHA-1 du commit vers lequel il pointe

=> Les branches sont donc faciles à créer et supprimer.

Aussi, comme chaque commit référence son parent, trouver la base de fusion appropriée lors d'un *merge* est également très simple



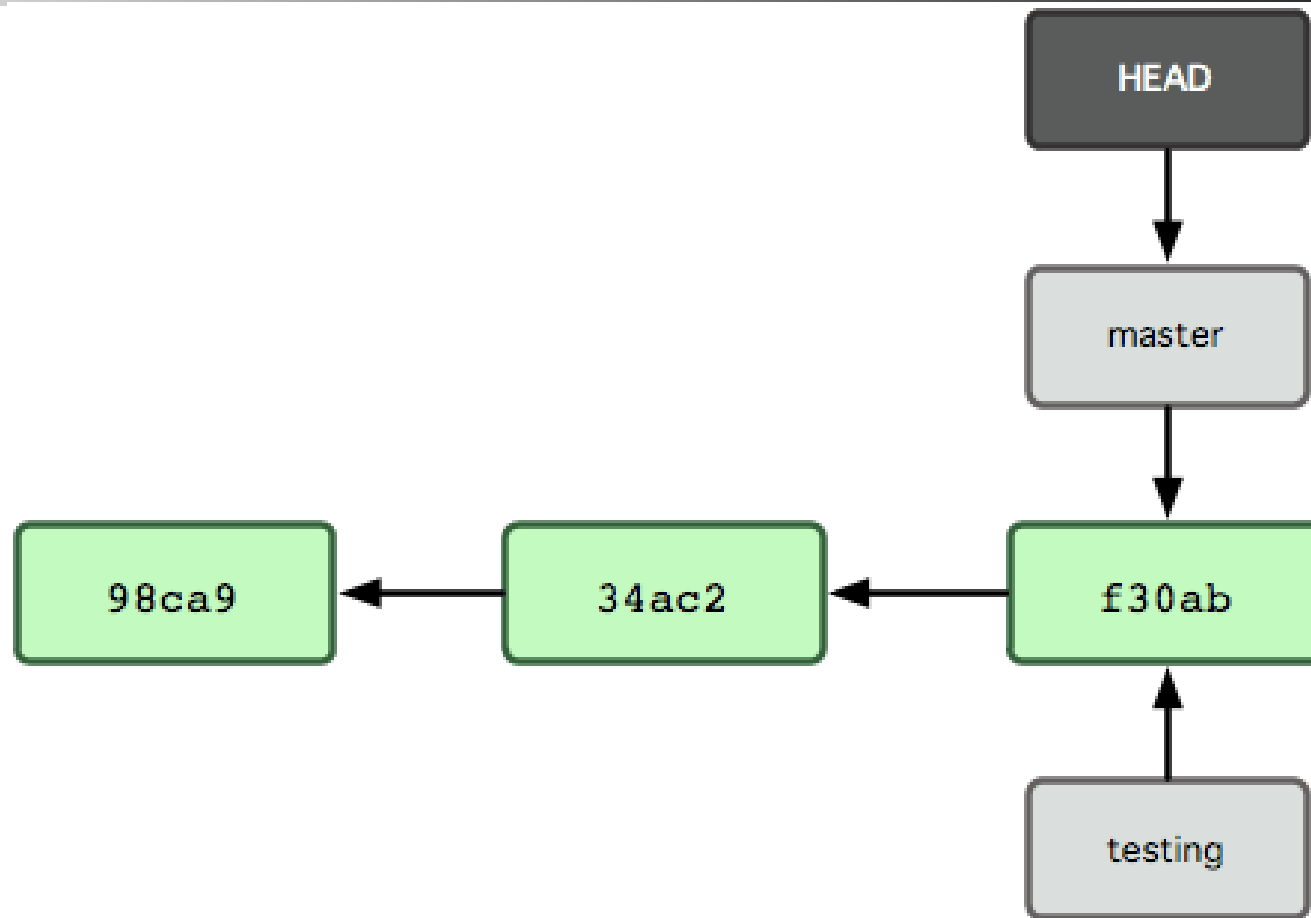
git checkout

Git connaît la branche courante sur laquelle on travaille en gardant un pointeur spécial nommé HEAD. (Pas de rapport avec le concept de HEAD de SVN ou CVS).

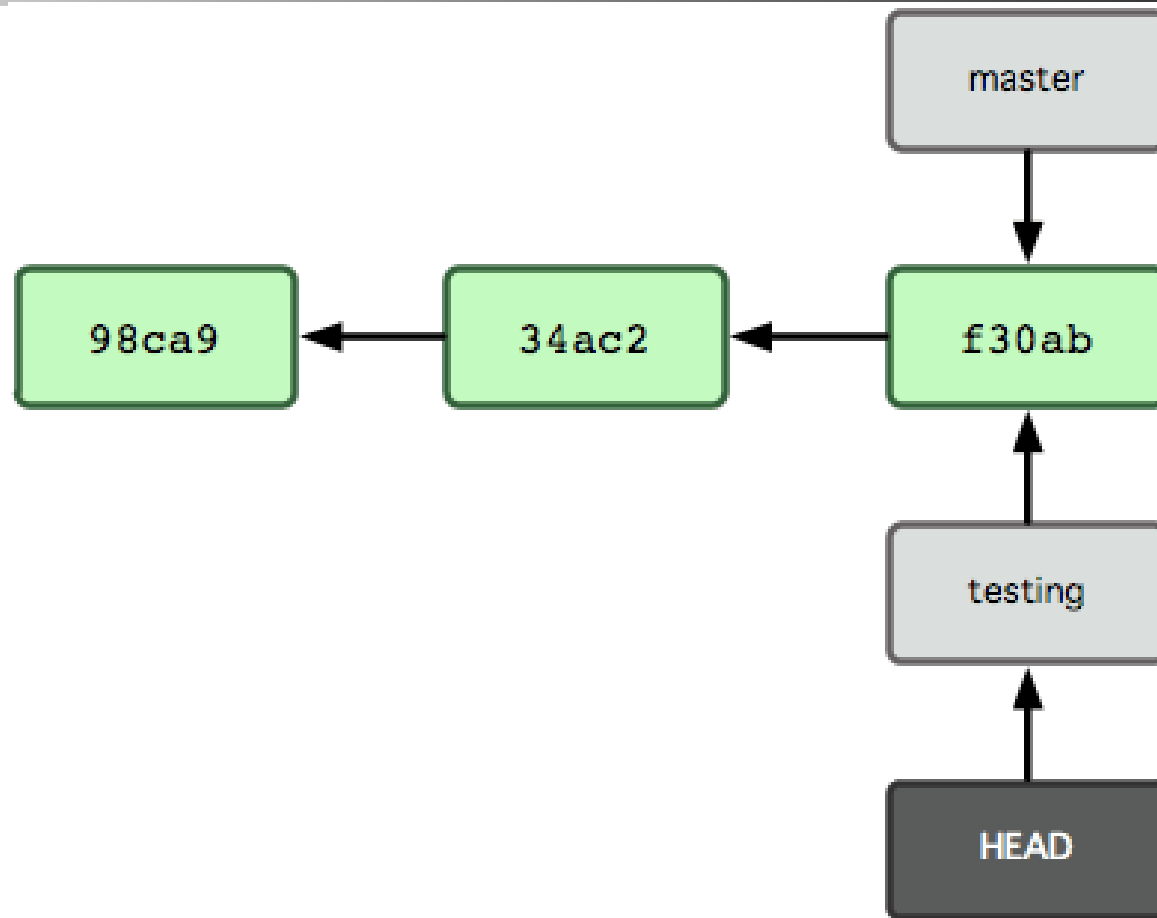
Pour basculer ce pointeur vers une branche existante, il faut exécuter la commande ***git checkout*** :

```
$ git checkout testing
```

Avant basculement



Après basculement

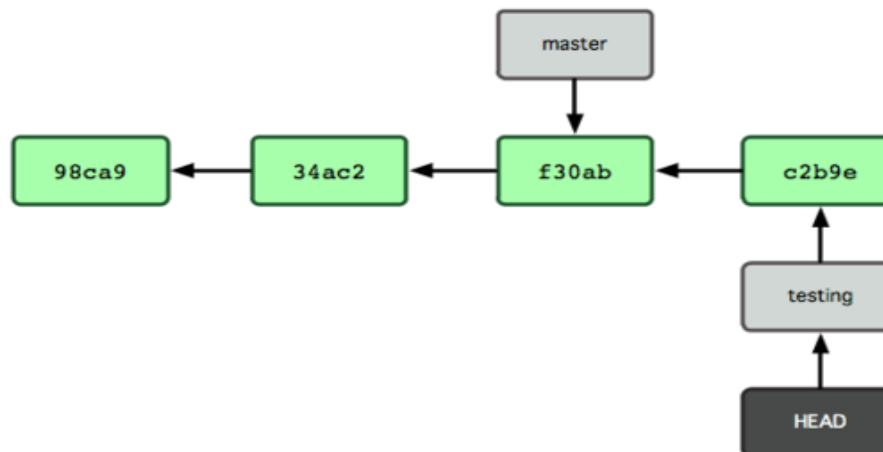


Commit sur une branche

Si on effectue un nouveau commit, ce sont les pointeurs *testing* et *HEAD* qui se déplacent, le pointeur *master* n'est pas modifié.

```
$ vim test.rb
```

```
$ git commit -a -m 'made a change'
```

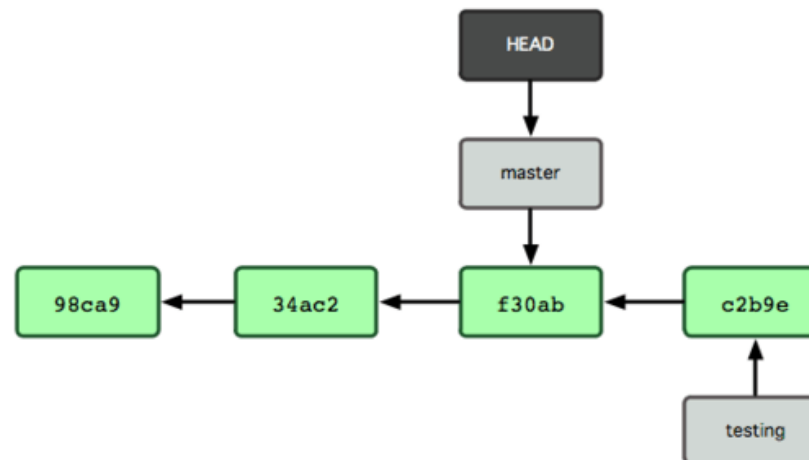


Retour à master

Pour revenir à la branche master :

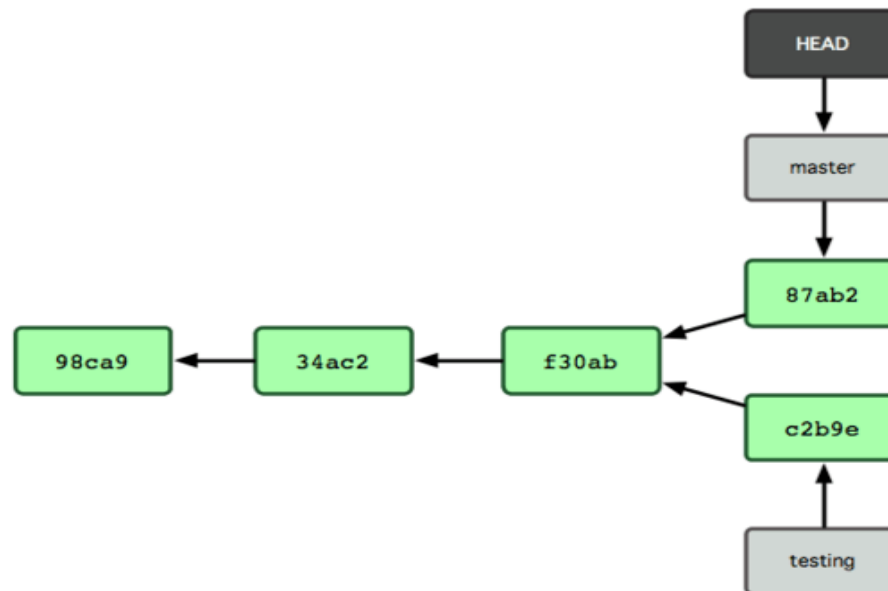
```
$ git checkout master
```

Le pointeur HEAD revient sur le pointeur master et le répertoire de travail est modifié pour reprendre l'instantané vers lequel le pointeur *master* pointe.



Divergence des branches

Si l'on effectue un nouveau commit, les branches divergent





Brancher et fusionner



Scénario

Une version du projet a été déployée en production. Vous décidez de démarrer une nouvelle version

- Une branche est créée pour le développement de la nouvelle version
- Le travail commence sur cette branche et des commits sont effectués

Lors du développement de la nouvelle version, un bug critique est détecté sur la version en production. Il faut absolument le corriger et déployer au plus vite

En tant que développeur, il faut

- Retourner à la branche de production
- Créer une branche pour travailler sur une correction du bug
- Lorsque la correction est effective, fusionner la branche corrective avec la branche de la production
- Retourner aux travaux de la nouvelle version



Création nouvelle branche

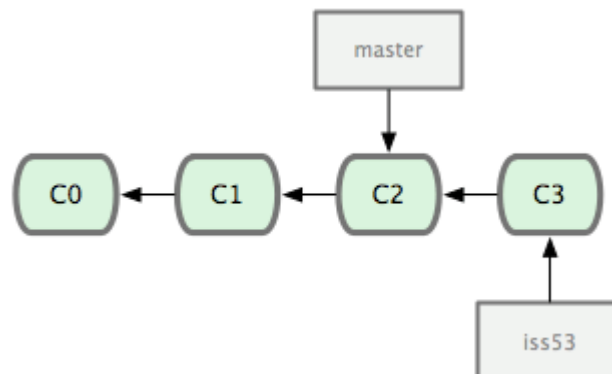
Pour développer la nouvelle version, on crée une nouvelle branche

=> création de branche et checkout

```
$ git checkout -b iss53
```

Switched to a new branch 'iss53'

Après quelques commits :





Basculement sur la branche de production

Premièrement, vous devez avoir un répertoire de travail propre avant de basculer de branche. Il faut donc committer (ou mettre de côté) le travail courant.

Ensuite basculer vers la branche de production :

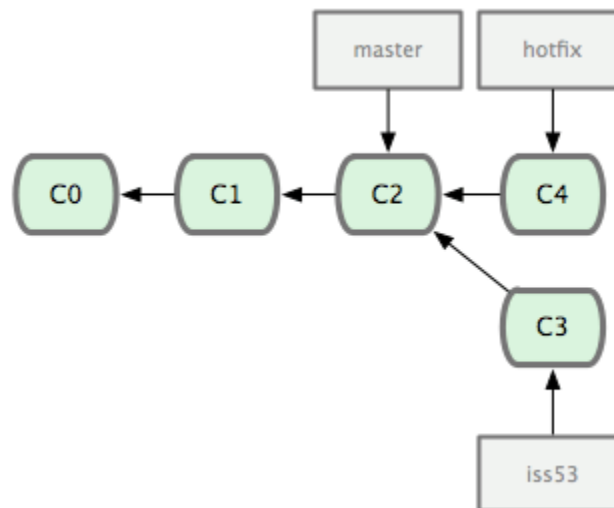
```
$ git checkout master  
Switched to branch 'master'
```

Le répertoire de travail revient à l'état de la production.

Correction du bug critique

Pour la correction du bug, il est conseillé de créer une branche afin de travailler sur cette branche jusqu'à ce que la correction soit terminée :

```
$ git checkout -b hotfix  
Switched to a new branch 'hotfix'  
$ vim index.html  
$ git commit -a -m 'fix the broken email address'  
[hotfix 3a0874c] fix the broken email address  
1 files changed, 1 deletion(-)
```





Fusion rapide

Une fois la correction effectuée, on peut la reporter dans la branche *master* avec la commande *merge*

```
$ git checkout master
```

```
$ git merge hotfix
```

```
Updating f42c576..3a0874c
```

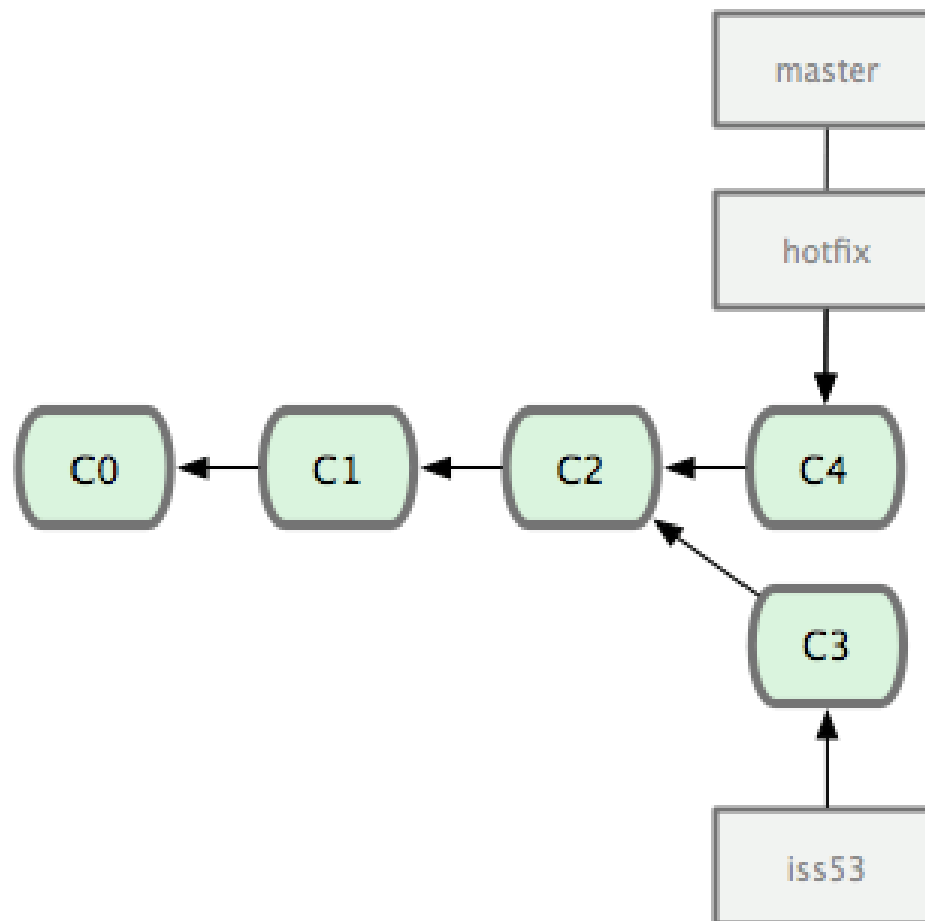
```
Fast-forward
```

```
README | 1 -
```

```
1 file changed, 1 deletion(-)
```

Dans ce cas, c'est une fusion **rapide** (*fast-forward*) car il n'y a pas de travaux divergents.

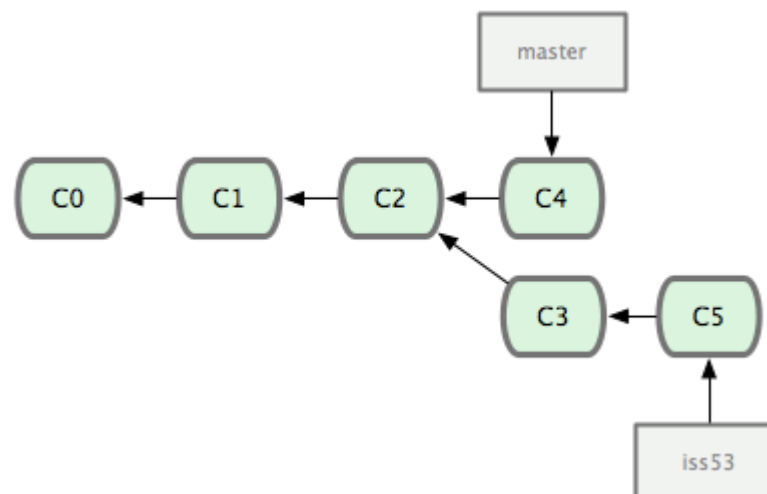
États après la fusion



Retour sur le développement

On peut supprimer la branche de correction, basculer vers la branche de développement et continuer de travailler

```
$ git branch -d hotfix
Deleted branch hotfix (was 3a0874c).
$ git checkout iss53
Switched to branch 'iss53'
$ vim index.html
$ git commit -a -m 'finish the new footer [issue 53]'
[iss53 ad82d7a] finish the new footer [issue 53]
1 file changed, 1 insertion(+)
```





Intégration de la correction

Il faut signaler que la correction effectuée n'est pas incorporée dans la branche de la nouvelle version

- Si nécessaire, il faut fusionner la branche *master* avec la branche de développement
- Sinon, attendre d'intégrer ses changements lorsque la branche de développement sera rapatriée dans la branche master



Fusion de la branche de développement

Une fois la nouvelle version prête, il est temps de fusionner la branche de développement avec la branche *master*

```
$ git checkout master
```

```
$ git merge iss53
```

```
Auto-merging README
```

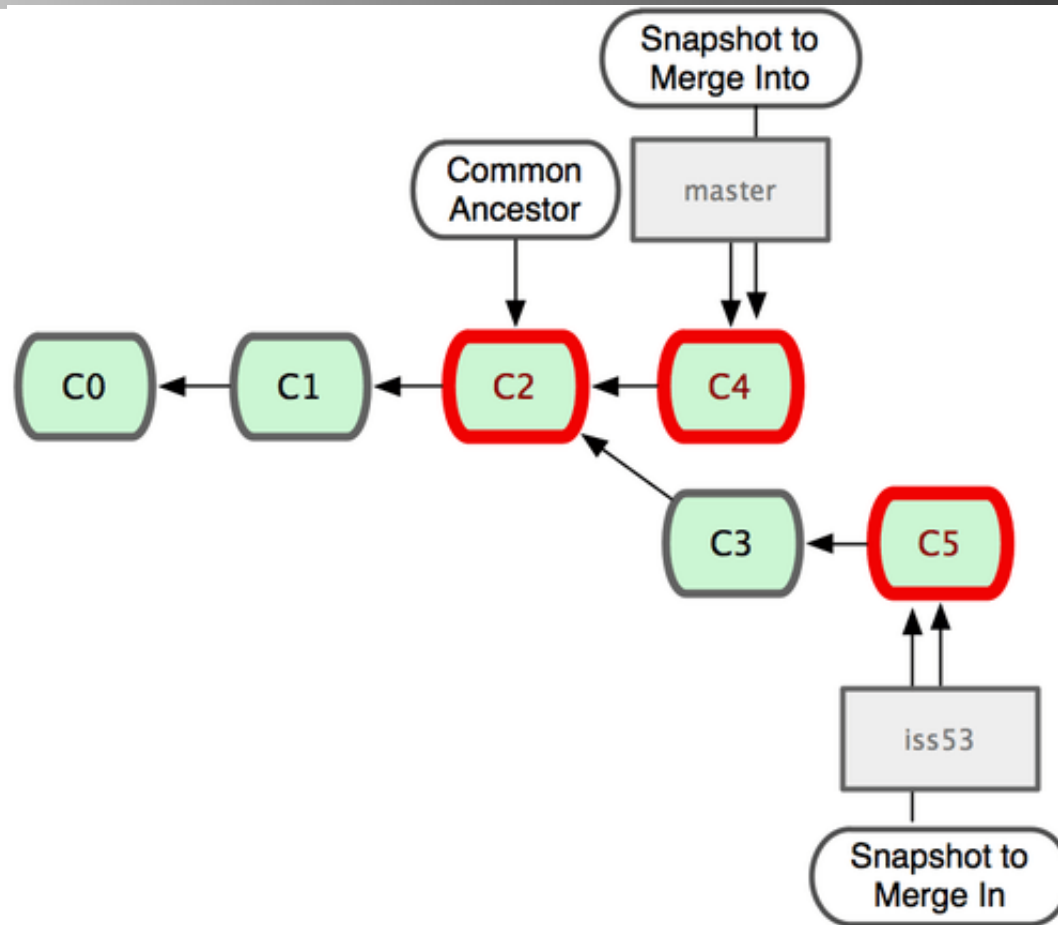
```
Merge made by the 'recursive' strategy.
```

```
README | 1 +
```

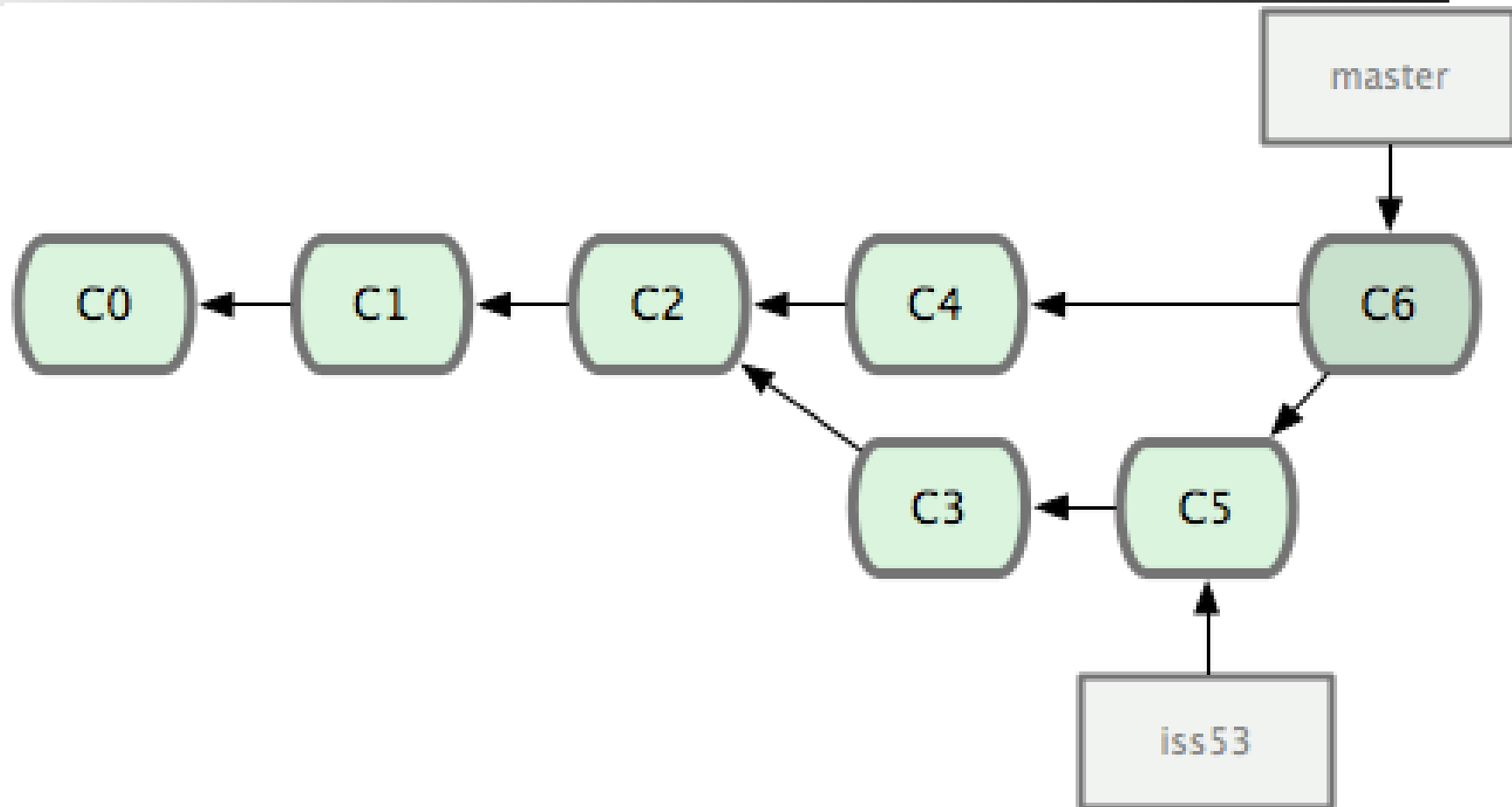
```
1 file changed, 1 insertion(+)
```

Dans ce cas les branches de développement et de production ayant divergé, Git doit effectuer une fusion en utilisant les 2 instantanés des différentes branches, ainsi que l'instantané de leur ancêtre commun (base de fusion)

Ancêtre commun



Résultat de la fusion





Conflits

Si les mêmes parties d'un fichier ont été modifiées différemment dans les 2 branches, Git n'est pas capable de faire la fusion lui-même et un conflit apparaît :

```
$ git merge iss53
```

```
Auto-merging index.html
```

```
CONFLICT (content): Merge conflict in index.html
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

Git hasn't automatically created a new merge commit. If you want to see which files are unmerged at any point after a merge conflict, you can run `git status`



Marqueurs

Git ajoute les marqueurs standard pour la résolution de conflits, il faut alors ouvrir manuellement les fichiers textes concernés pour supprimer ses marqueurs

```
<<<<<< HEAD
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53
```



Résolution

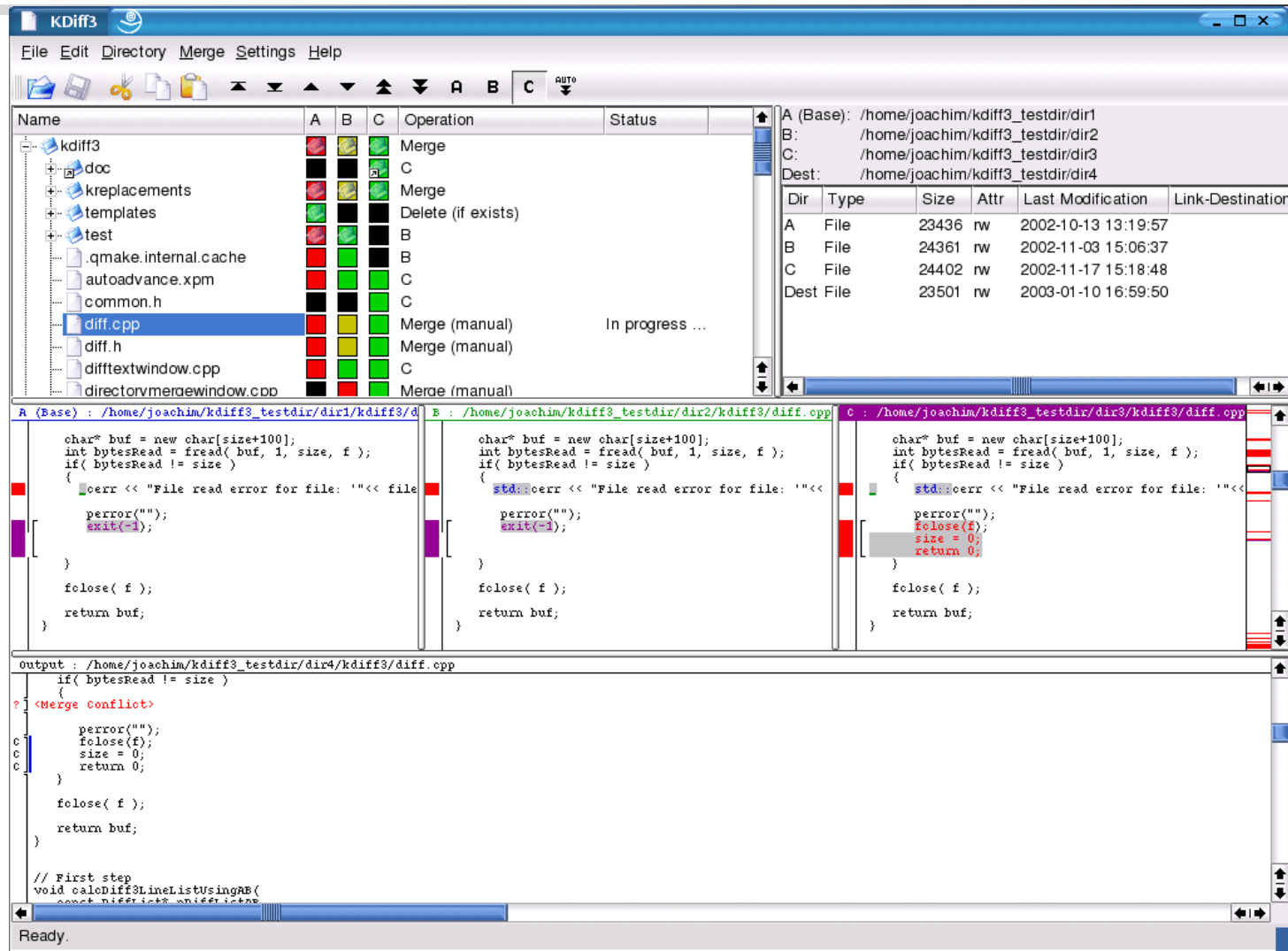
La résolution consiste donc généralement à choisir une des 2 modifications ou la fusion des 2. Par exemple :

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Après avoir résolu chaque section en conflits, il faut exécuter **git add** pour ajouter les fichiers dans l'index. Git les considère alors résolus.

Il est également possible d'exécuter **git mergetool** qui démarre un outil graphique configuré pour la résolution de conflit.

Exemple *KDiff*





Commit final

A la fin, pour finaliser la fusion il faut committer :

```
$ git commit
```

```
Merge branch 'iss53'
```

```
Conflicts:
```

```
    index.html
```

```
#
```

```
# It looks like you may be committing a merge.
```

```
# If this is not correct, please remove the file
```

```
#       .git/MERGE_HEAD
```

```
# and try again.
```

```
#
```



Commande *git branch*

La commande ***git branch*** sans argument liste les branches courantes et celle qui est actuellement utilisée

```
$ git branch
```

```
iss53
```

```
* master
```

```
testing
```

L'option ***-v*** permet de visualiser le dernier commit de chaque branche

L'option ***--merged*** permet de voir les branches déjà fusionnées

L'option ***--no-merged*** permet de voir toutes les branches contenant des travaux non fusionnés



Exemples

```
$ git branch -v
```

```
prob53    93b412c fix javascript issue
```

```
* master   7a98805 Merge branch 'prob53'
```

```
test      782fd34 add scott to the author list
```

```
$ git branch --merged
```

```
prob53
```

```
* master
```

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

```
test
```



Rebaser



Introduction

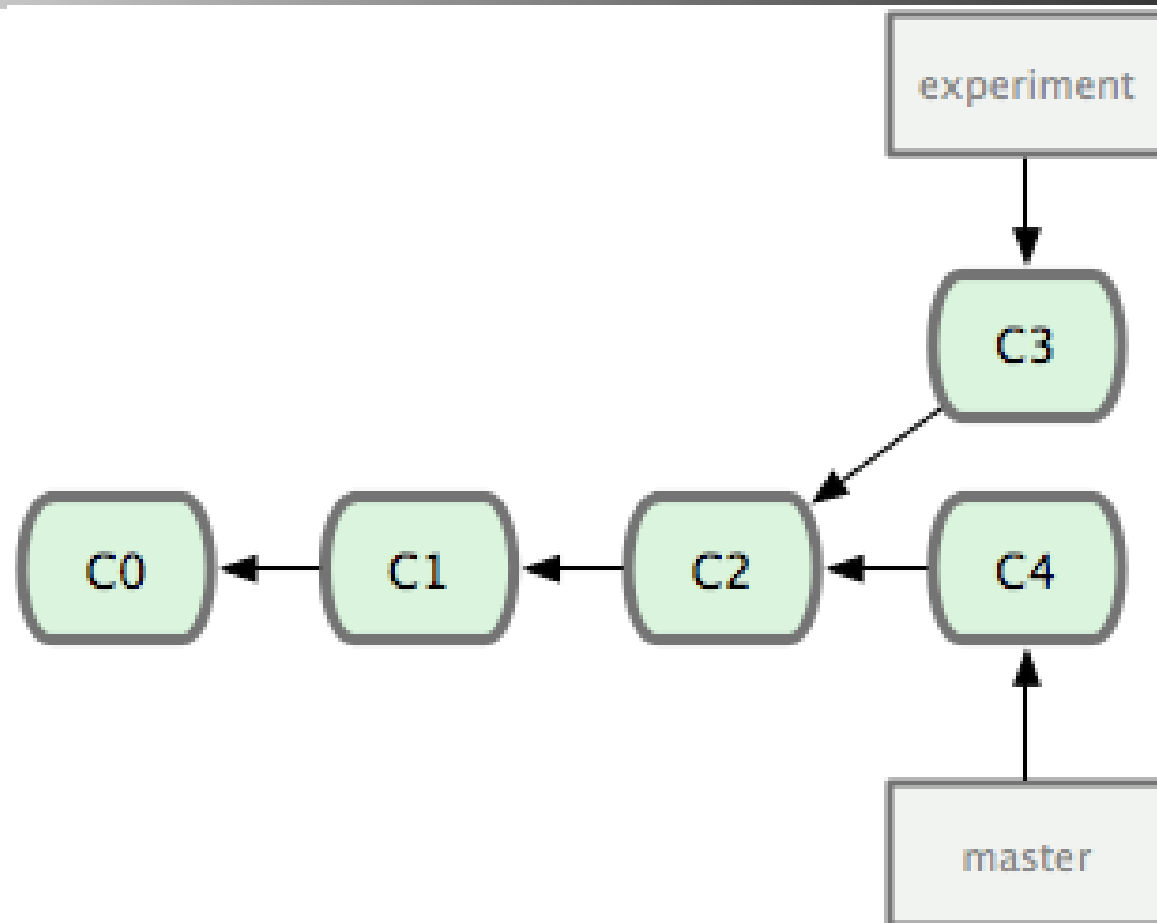
Il y a deux façons d'intégrer les modifications d'une branche dans une autre :

- **merge** : Joindre et fusionner les 2 têtes d'une ligne de commit
- **rebase** : Rejouer les modifications d'une ligne de commits sur une autre dans l'ordre d'apparition

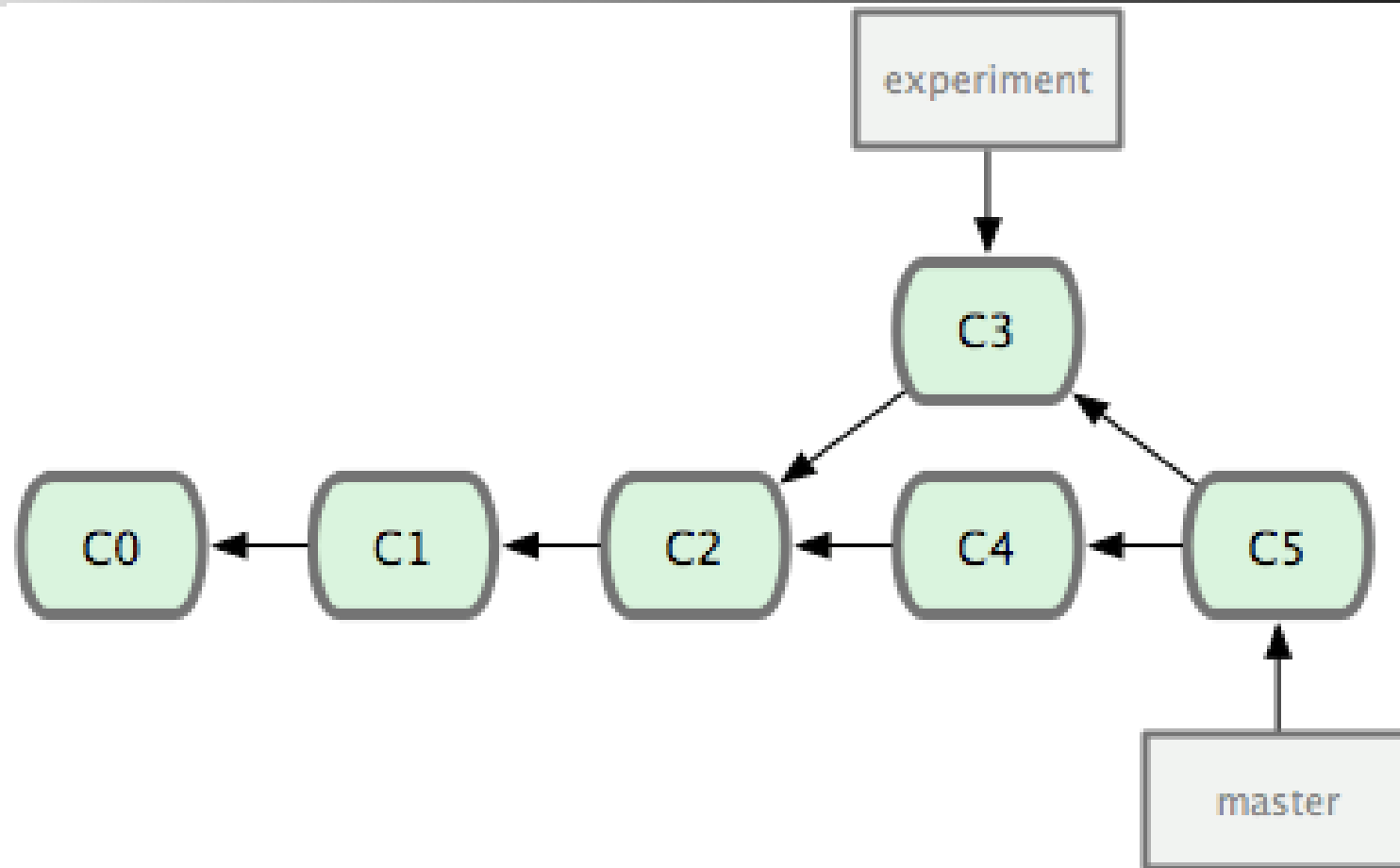
rebaser rend l'historique plus clair et permet de s'assurer que des patches s'appliquent correctement sur une branche distante

=> Le travail du mainteneur de projet est facilité

2 branches divergentes



Fusion





Rebase

L'opération de rebase prend le patch de la modification introduite en *C3* et le réapplique sur *C4*.

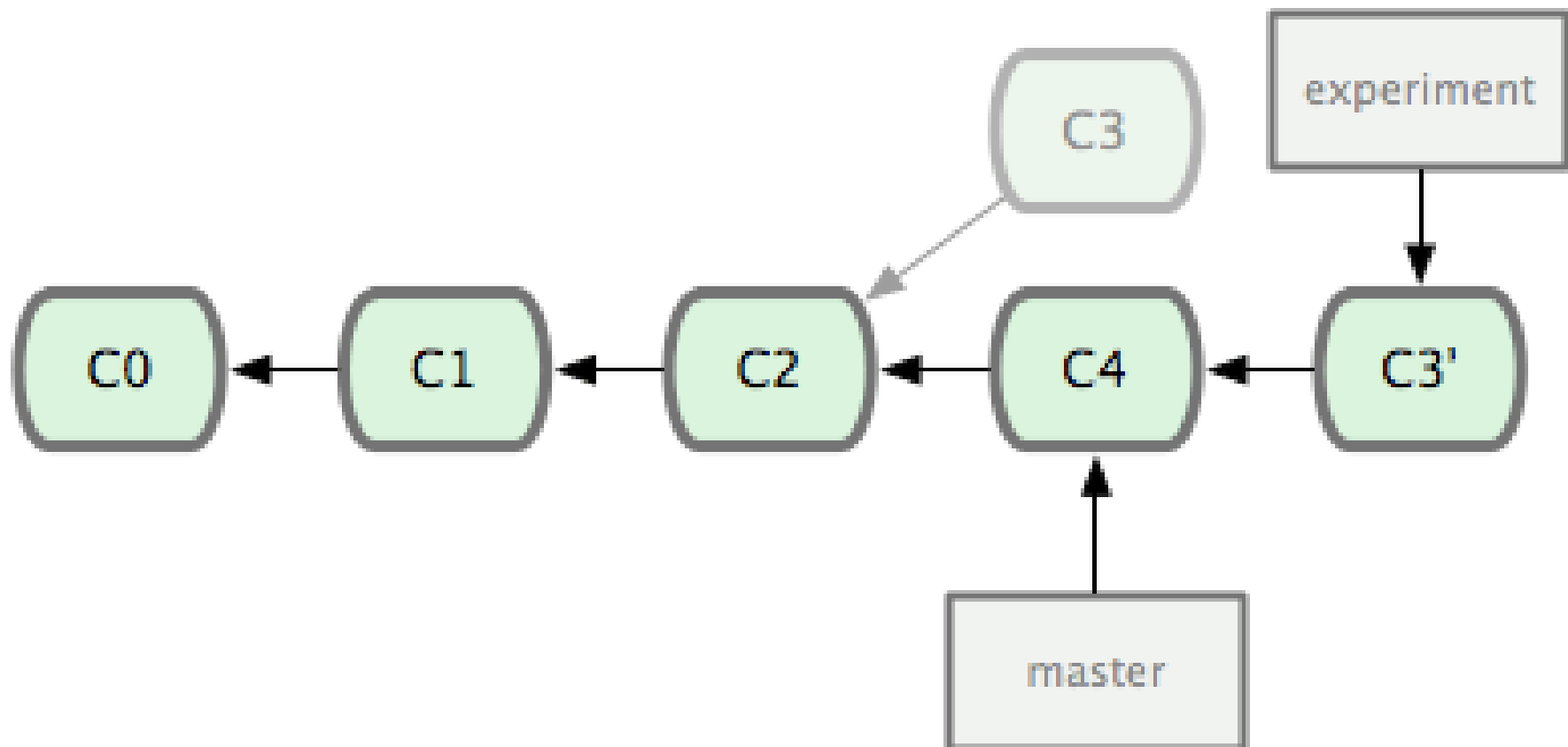
```
$ git checkout experience
```

```
$ git rebase master
```

```
First, rewinding head to replay your work on  
top of it...
```

```
Applying: added staged command
```

Rebase





Rebasing et conflit

Si un conflit apparaît lors de l'application d'un patch particulier, l'opération de rebasing s'interrompt

Il faut alors soit :

- Résoudre le conflit et continuer l'opération de rebasing
`git add` après la résolution du conflit
`git rebase --continue` pour continuer le rebasing
- Ignorer l'application de ce patch
`git rebase --skip`
- Arrêter l'opération de rebasing
`git rebase --abort`



Typologie des branches

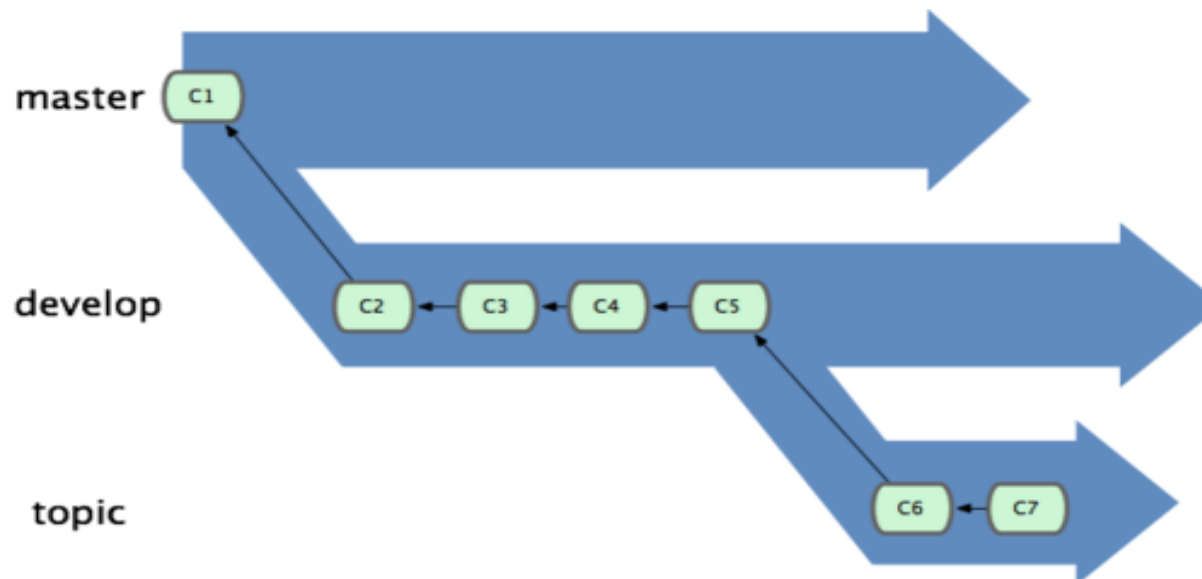
Branches longues

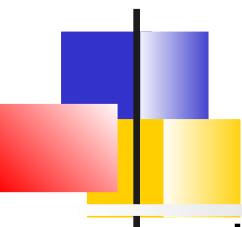
Fusionner une branche dans une autre plusieurs fois sur une longue période est généralement facile.

Cela signifie que l'on peut avoir plusieurs branches ouvertes correspondantes à des étapes du développement et des niveaux de stabilité

Lorsqu'une branche atteint un niveau plus stable, elle est alors fusionnée avec la branche d'au-dessus.

Ce type d'organisation est assez utile pour les gros projets.





Branches thématiques

Les branches thématiques sont utiles pour tout type de projet.

Ce sont des branches généralement **locales**, de **courte durée** créées pour une fonctionnalité ou une tâche **particulière**

- La simplicité des opérations sur les branches de Git favorise ce type d'organisation

L'avantage de cette approche est de séparer les travaux en silos (à chaque fonctionnalité est associé un ensemble de changements) et donc de faciliter la revue de code

Les changements peuvent être fusionnés lorsqu'il sont prêts (minutes, jours ou jamais) indépendamment de l'ordre dans lequel ils ont été développés.



Tags



Introduction

Comme dans la plupart des SCMs, Git permet de **tagger** certains commits

Généralement, cela est utilisé pour marquer les releases (v1.0, etc)

Les commandes associées aux tags permettent de lister les tags disponibles et leurs types, créer de nouveaux tags, etc.



Lister les tags

La commande ***git tag*** liste les tags dans l'ordre alphabétique.

```
$ git tag
```

```
v0.1
```

```
v1.3
```

Il est possible de filtrer la sortie avec l'option ***-l***

```
$ git tag -l 'v1.4.2.*'
```

```
v1.4.2.1
```

```
v1.4.2.2
```

```
v1.4.2.3
```

```
v1.4.2.4
```



Types de tags

Git utilise 2 types de tags :

- Un tag **simple** (équivalent à une branche qui n'évolue pas) est juste un pointeur vers un commit particulier
- Un tag **annoté**, stocké comme un objet complet dans la base Git, contient un checksum, le nom de la personne qui a taggé, son email, la date et un message de tag.
Un tag annoté peut également être signé et vérifié avec GNU Privacy Guard (GPG)



Création d'un tag annoté

L'option **-a** de la commande *git tag* permet d'ajouter un tag annoté.

Un message est alors nécessaire soit via l'éditeur soit en ligne via l'option *-m*

```
$ git tag -a v1.4 -m 'my version 1.4'
```

```
$ git tag
```

```
v0.1
```

```
v1.3
```

```
v1.4
```



Afficher les informations d'un tag

La commande ***git show*** permet de revoir les informations associées à un tag

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```



Tags signés

L'option **-s** associée à une clé privée précédemment créée avec GPG (*Gnu Privacy Guard*) permet de signer un tag

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
```

```
You need a passphrase to unlock the secret key for  
user: "Scott Chacon <schacon@gee-mail.com>"
```

```
1024-bit DSA key, ID F721C45A, created 2009-02-09
```



Vérifier un tag

Pour vérifier un tag signé, l'option **-v** est utilisée. La commande utilise alors GPG et la clé publique du signataire pour vérifier le tag

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700
```

```
GIT 1.4.2.1
```

```
Minor fixes since 1.4.2, including git-mv and git-http with alternates.
```

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
```

```
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
```

```
gpg:                aka "[jpeg image of size 1513]"
```

```
Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A
```



Tags simples

Pour créer un tag simple, il ne faut pas utiliser les options *-a*, *-s*, ou *-m* :

```
$ git tag v1.4-lw
```

```
$ git tag
```

```
v0.1
```

```
v1.3
```

```
v1.4
```

```
v1.4-lw
```

```
v1.5
```



Tagger à posteriori

Il est possible de tagger un commit à posteriori en indiquant tout simplement le checksum à la commande.

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```



Le serveur Gitlab

Installation
Projets et membres
Gestion des issues
Dépôts Gitlab



Introduction GitLab

Gitlab est devenu un outil de gestion d'un cycle de vie **DevOps**

- Interface web gérant des référentiels Git et fournissant des fonctionnalités de collaboration, de suivi des problèmes et de pipeline CI / CD
- S'appuie sur les commandes de bases de Git
- S'intègre avec d'autres produits (annuaire LDAP, *Jira*, *Mattermost*, *Kubernetes*, *Slack* par exemple)
- Disponible sous une édition communautaire et entreprise



Installations

Gitlab s'installe sous Linux. Différentes façons :

- **Omnibus Gitlab** : Packages pour différentes distributions de Linux
- **GitLab Helm chart** : Version Cloud, installation sous Kubernetes
- Images **Docker**
- A partir des **sources**

Également disponible en ligne : *gitlab.com*



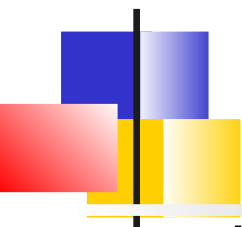
Community vs Enterprise

Le même cœur, l'*enterprise edition* ajoute du code propriétaire.

Il est possible d'utiliser *Enterprise* sans payer => Idem en fonctionnalités que l'édition communautaire

Les versions payantes apportent :

- Plus de fonctionnalités relatives aux Issues :
- Recherche de code avancé via Elastic Search, Revue de code visuel,
- Intégration LDAP, Kerberos, JIRA, Jenkins. Emailing
- Dépôts *Maven*, *npm* et *docker*
- Dépôts miroir distants
- PostgreSQL HA ...
- Support 24h/24



Projets

Un projet *Gitlab* est associé à un dépôt Git

Par défaut, tous les utilisateurs *Gitlab* peuvent créer un projet

3 visibilité sont possibles :

- **Public** : Le projet peut être cloné sans authentification.
Tout utilisateur a la permission *Guest*
- **Interne** : Peut être cloné par tout utilisateur authentifié.
Tout utilisateur a la permission *Guest*
- **Privé** : Ne peut être cloné et visible seulement par ses membres



Fonctionnalités

Un projet apporte plusieurs fonctionnalités :

- **Gestion de dépôts** : Visualisation des sources, des branches, des tags, des historique, de l'activité, édition des sources
- **Suivi d'issues** : Collaboration sur le travail planifié, milestones, Tableau de bords
- **Merge Request** : Modifications en cours du code source
- **Pipelines de CI/CD** : Constructions, Tests et déploiements automatisés sur le code source
- **Autres** : Wiki, Tableaux de bords, Historique des déploiements sur les différents environnements, Gestion de release, Dépôts d'artefacts,

Menus

Projects : Informations sur les commits, les branches, l'activité, les releases, tdb sur la productivité

Repository : Navigateur de fichiers, Commits, branches, tags, historique, comparaison, statistiques sur les fichiers du projet

Issues : Gestion des issues, tableau de bord Kanban

Merge requests : Travaux en cours

CI/CD : Historique d'exécution des pipelines

Operations : Gestion des environnements de déploiement

Packages : Accès au registre de conteneur

Wiki : Documentation annexe

Snippets : Bouts de code

Settings : Configuration projet, Visibilité, Merge Request, Membres, pipeline, intégration avec d'autres outils



Membres

Les utilisateurs peuvent être affectés à des projets, ils en deviennent **membres**

Un membre a un rôle qui lui donne des permissions sur le projet :

- **Guest** : Créer un ticket
- **Reporter** : Obtenir le code source
- **Developer** : Push/Merge/Delete sur les branches non protégée, Merge request sur les autres branches
- **Maintainer** : Administration de l'équipe, Gestion des branches protégés ou non, Tags, Ajouts de clés SSH
- **Owner** : Créateur du projet, a le droit de le supprimer



Groupes de projets

Afin de faciliter la gestion des membres et de leurs permissions, il est possible de définir des **groupes de projets**.

- Les membres et leurs rôles sont donc définis au niveau du groupe
=> Ils ont alors accès à tous les projets du groupe

Les groupes peuvent être hiérarchiques

Attention : Il est dangereux de déplacer un projet existant dans un autre groupe

Settings -> General -> Advanced -> Transfer project -> Select a new namespace



Share with group

Si les membres du groupe
« *Engineering* » doivent avoir accès à
un autre projet appartenant déjà à un
autre groupe, il faut utiliser la fonction
« ***Share with group*** »

Sur l'autre projet :

Settings → Members → Share with group



Configuration Utilisateur

Dans la partie *Settings* d'un utilisateur, en dehors des informations personnelles, on retrouve :

- La configuration des notifications par projet
- La gestion des clés SSH facilitant l'authentification
- La gestion des clés GPG permettant de signer des tags
- Les préférences (en particulier la langue)



Mise en place clés ssh

La mise en place des clés *ssh* permet de pouvoir interagir avec Gitlab sans avoir à fournir de mot de passe.

2 étapes :

- Créer une paire de clé privé/publique
- Fournir la clé publique à Gitlab via l'interface web



Mise en place

- Environnement Linux :

```
ssh-keygen -t ed25519 -C "email@example.com"
```

Ou

```
ssh-keygen -o -t rsa -b 4096 -C "email@example.com"
```

- Copier le contenu de la clé publique (*.pub) dans l'interface Gitlab
- Tester avec :

```
ssh -T git@gitlab.com
```



Gestion des issues



Introduction

Les **Issues** permettent la collaboration avant et pendant leur implémentation

Elles peuvent être utilisées pour différents cas d'utilisation :

- Discuter de l'implémentation d'une nouvelle idée
- Suivi de tâches
- Backlog agile, Reporting de bug, Demande de support

Elles sont toujours associées à un projet.

Elles peuvent être visualisées par groupe de projets.



Données associées à une *issue*

Contenu :

- Titre
- Description et tâches
- Commentaires et activité

Membres

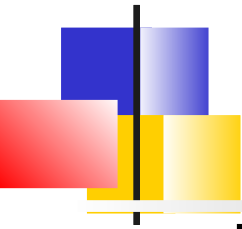
- Auteur
- Responsables

Etat

- Status (ouvert/fermé)
- Confidentialité
- Tâches (terminé ou en suspens)

Planning et suivi

- Milestone
- Date de livraison
- Poids
- Suivi du temps
- Tags (Labels)
- Votes
- Reaction emoji
- Issues liées
- Epic (collection d'issues) affectée
- Identifiant et URL

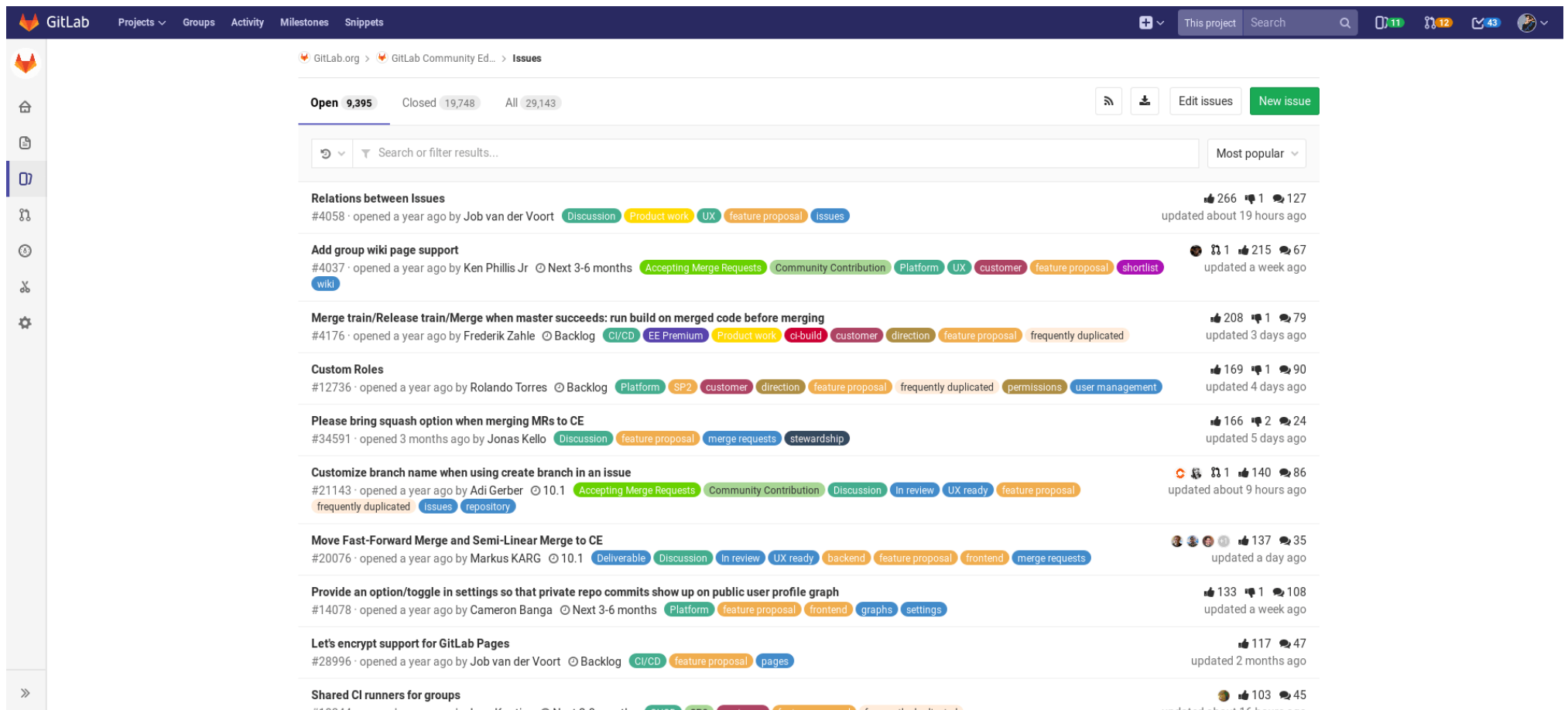


Visualisation des issues

Les issues peuvent être visualisées via :

- Une **liste**. Elle affiche toutes les issues du projet ou de plusieurs projets. On peut les filtrer ou faire des actions par lots (bulk)
- Le **tableau de bord Kanban** qui affiche des colonnes en fonction des labels (par défaut statut de l'issue) ou des responsables. Les workflows sont customisable via les labels
- **Epic** : Vision transversale aux projets des issues partageant un thème, un milestone,

Liste

 GitLab

Projects ▾Groups ▾Activity ▾Milestones ▾Snippets ▾

+

▼

This project

Search

Q

11

12

43

GitLab.org > GitLab Community Ed... > Issues

Open 9,395Closed 19,748All 29,143

Search or filter results...

Most popular ▾

Relations between Issues

#4058 · opened a year ago by Job van der Voort

DiscussionProduct workUXfeature proposalIssues

2661127

updated about 19 hours ago

Add group wiki page support

#4037 · opened a year ago by Ken Phillis Jr

Next 3-6 monthsAccepting Merge RequestsCommunity ContributionPlatformUXcustomerfeature proposalshortlistwiki

1121567

updated a week ago

Merge train/Release train/Merge when master succeeds: run build on merged code before merging

#4176 · opened a year ago by Frederik Zahle

BacklogCI/CDEE PremiumProduct workci-buildcustomerdirectionfeature proposalfrequently duplicated

208179

updated 3 days ago

Custom Roles

#12736 · opened a year ago by Rolando Torres

BacklogPlatformSP2customerdirectionfeature proposalfrequently duplicatedpermissionsuser management

169190

updated 4 days ago

Please bring squash option when merging MRs to CE

#34591 · opened 3 months ago by Jonas Kello

Discussionfeature proposalmerge requestsstewardship

166224

updated 5 days ago

Customize branch name when using create branch in an issue

#21143 · opened a year ago by Adi Gerber

10.1Accepting Merge RequestsCommunity ContributionDiscussionIn reviewUX readyfeature proposalfrequently duplicatedIssuesrepository

14086

updated about 9 hours ago

Move Fast-Forward Merge and Semi-Linear Merge to CE

#20076 · opened a year ago by Markus KARG

10.1DeliverableDiscussionIn reviewUX readybackendfeature proposalfrontendmerge requests

13735

updated a day ago

Provide an option/toggle in settings so that private repo commits show up on public user profile graph

#14078 · opened a year ago by Cameron Banga

Next 3-6 monthsPlatformfeature proposalfrontendgraphssettings

1331108

updated a week ago

Let's encrypt support for GitLab Pages

#28996 · opened a year ago by Job van der Voort

BacklogCI/CDfeature proposalpages

11747

updated 2 months ago

Shared CI runners for groups

#10244 · opened a year ago by Ivan Kertész

Next 3-6 monthsCI/CDSP2customerfeature proposalfrequently duplicated

10345

updated about 16 hours ago

Kanban

The screenshot displays a GitLab Kanban board for the 'Development' branch. The board is organized into three main columns: 'In dev', 'In review', and 'Closed'. Each column has a header with a count of items and a 'View scope' button. The 'In dev' column contains three cards, the 'In review' column contains two, and the 'Closed' column contains three. Each card represents a task or feature, with labels indicating its status and priority. A tooltip is visible over the first card in the 'In dev' column, providing additional context about the task.

Development Search or filter results... View scope

In dev 80 174

Read git data via gitaly
In dev Plan backend
Work for the Plan team. Covers Issues, Labels, Milestones, Boards, and more. See <https://about.gitlab.com/handbook/product/categories/>. Ping @victorwu for questions and comments.
line in merge
P1
code review devops:create direction
feature proposal frontend merge requests
gitlab-org/gitlab-ce#18008 5

In review 34 95

`ExpireBuildArtifactsWorker` is broken
In review P3 S3 Verify database
devops:verify missed-deliverable performance
gitlab-org/gitlab-ce#41057

Ensure that all CI/CD queries take less than 15 seconds to complete
Accepting merge requests In review Stretch
Verify database devops:verify meta
missed-deliverable performance
gitlab-org/gitlab-ce#40524

Closed 47352 19838

include brand ai styles
To Do
gitlab-org/design.gitlab.com#5

static page example
To Do
gitlab-org/design.gitlab.com#4

welcome page
To Do
gitlab-org/design.gitlab.com#3

add some real components
To Do
gitlab-org/design.gitlab.com#2

Multiple blocking merge request approval rules
Create Deliverable GitLab Premium In dev
P1 UX approvals backend customer
customer+ devops:create direction
feature proposal frontend merge requests

Further improvements to Project overview UI
Deliverable In review Manage P2
UX ready devops:manage direction frontend
missed-deliverable missed:11.5 project
release post item

Vue détaillée issue

The screenshot displays a GitLab issue page for 'Issue #1395' opened by Marcia Ramos. The page is annotated with numbers 1 through 18, highlighting various UI elements and features. The issue title is 'GitLab Issue 12' and the description is 'Hello World!'. The issue is marked as 'Open' and has 0 of 2 tasks completed. The right sidebar shows the 'Todo' list with items like '3 Assignees', 'Milestone 9.2', 'Time tracking', 'Due date', 'Labels', 'Weight', and '3 participants'. The bottom section shows a 'Write' comment box and a 'Comment' button. A tooltip for 'Create a merge request' is visible over the 'Create a merge request' button.

1 Open Issue #1395 opened about 14 hours ago by Marcia Ramos 0 of 2 tasks completed

2 New issue Close issue Edit

3 GitLab Issue 12

4 Hello World! 🍌

5 This is my issue's description, written in markdown (GitLab Flavored Markdown).

6 This is an `<h3>`

7 Let's quote someone here

8 Add a task list:

9 Task 1 Task 2

10 Mention merge requests (gitlab-org/gitlab-ee!1784 (merged)) and issues (gitlab-org/gitlab-ee#2101 (closed)) and hover over them to see their titles.

11 Invite users to collaborate with @mentions: @marcia

12 Edited just now by Marcia Ramos

13 1 Related Merge Request

14 !1784 How to Configure LDAP with GitLab EE in GitLab.org / GitLab Enterprise Edition Merged

15 0 0 0

16 Marcia Ramos @marcia assigned to @axil and @jivanvl about 14 hours ago

17 Marcia Ramos @marcia changed milestone to 9.2 about 14 hours ago

18 Marcia Ramos @marcia added on it label about 14 hours ago

19 Marcia Ramos @marcia changed time estimate to 30m about 14 hours ago

20 Create a merge request

21 Create a branch

22 Write Preview

23 Write a comment or drag your files here...

24 Markdown and slash commands are supported

25 Attach a file

26 Comment Close issue



Actions sur une issue (1)

1. Création, Fermeture, Edition des champs de base
2. Ajouter à sa Todo List, la marquer comme terminé
3. Responsable(s) de l'issue, peut être changé à tout moment
4. Affecter une issue à un milestone
5. Temps estimé, temps passé
6. Date de livraison, peut être changée à tout moment
7. Labels. Catégorise les issues et permet de mettre en place des workflows personnalisés reflétés dans le Kanban
8. Poids. Indicateur sur l'effort nécessaire associé à l'issue



Actions sur une issue

- 9. Participants. Indiqués dans la description ou qui ont participé à la discussion
- 10. Notifications. Permet de s'abonner/désabonner
- 11. Référence. Permet de copier l'URL d'accès
- 12. Titre et description (markup)
- 13. Mentions. Met en surbrillance pour la repérer facilement
- 14. Merge requests associés
- 15. emoji
- 16. Thread. Commentaires organisés en threads



Autres fonctionnalités

Issues liées : Permet d'associer une issue à une autre (Travail préliminaire, contexte, dépendance, doublon)

Crosslinking : Liens vers des objets référençant l'issue.
(Commit, Autre Issue ou Merge Request)

- Par exemple un commit

- `git commit -m "this is my commit message. Ref #xxx"`

Fermeture automatique : Possibilité de fermer les issues automatiquement après un merge request

Gabarits : Créer des issues à partir de gabarits

Edition bulk

Import/Export d'issues

API Issues



Labels

Ils permettent de catégoriser les issues ou MR.
Par exemple : *bug*, *feature request*, ou *docs*.

- Chaque label a une couleur personnalisable.

Les **scoped labels** ont un format *clé :: valeur*.

- A un instant *t*, une issue ne peut pas avoir plusieurs labels de la même clé.
- Cela peut permettre de définir des workflows.

Exemple de scoped labels

workflow::development,

workflow::review

workflow::deployed



Milestones

Ils permettent d'organiser les issues et MR dans un groupe cohérent, avec une date de début et une date d'échéance (facultatives).

Ils peuvent définir :

- des sprints Agile
- des releases

Ils peuvent être associés à des groupes



Dépôts Gitlab



Particularités *Gitlab*

On peut interagir avec les dépôts GitLab via **l'UI** ou en **ligne de commande**.

GitLab supporte des langages de **markup** pour les fichiers du dépôt. Utilisé principalement pour la documentation

Lorsqu'un fichier **README** ou index est présent, son contenu est immédiatement rendu (sans ouverture du fichier)

L'UI donne la possibilité de **télécharger** le code source et les archives générées par les pipelines

Verrouillage de fichier : Empêcher qu'un autre fasse des modifications sur le fichier pour éviter des conflits.

Accès aux données via **API**. Exemple :

```
GET /projects/:id/repository/tree
```



Particularités du commit

- **Skip pipelines**: Si le mot-clé **[ci skip]** est présent dans le commit, la pipeline de GitLab ne s'exécute pas.
- **Cross-link issues/MR**: Si on mentionne une issue ou un MR dans un message de commit, ils seront affichés sur leur thread respectif.
- Il est possible via l'UI d'effectuer aisément un *cherry-pick* ou un *revert* d'un commit particulier
- Possibilité de signer les commits via GPG



Vues proposées

Settings → *Contributors* : Les contributeurs au code

Repository → *Commits* : Historique des commits

Repository → *Branches/Tags* : Gestion des branches et des tags

Repository → *Graph* : Vue graphique des commits et merge

Repository → *Charts* : Affiche les langages détectés par Gitlab et des statistiques sur des commits



Branche par défaut

A la création de projet, *GitLab* positionne *master* comme branche par défaut.

Peut-être changé *Settings* → *Repository*.

C'est dans la branche par défaut que sont fusionnées les modifications relatives à une issue lors d'un *merge request*.

La branche par défaut est également une branche protégée



Création de branche

Plusieurs façons de **créer des branches** avec Gitlab :

- A partir d'une issue, la branche est donc documentée avec la collaboration sur l'issue
- A partir du tableau de bord projet, de la même façon la branche sera fusionnée dans la branche par défaut



Branche protégée

Un branche protégée

- Seul un membre avec au moins la permission *Maintainer* peut la créer
- Seul un *Maintainer* peut y faire des push
- Il empêche quiconque de forcer un push vers la branche
- Il empêche quiconque de supprimer la branche

On peut utiliser des *wildcards* pour protéger plusieurs branches en même temps. *Ex :*

-stable, production/



Permissions pour « push » et « merge »

Les permissions par défaut d'une branche protégée peuvent être surchargées avec les champs de configuration “**Allowed to push**” et “**Allowed to merge**”

Par exemple, on peut positionner

- “*Allowed to push*” à “*No one*”
- “*Allowed to merge*” à “*Developers + Maintainers*”

=> Tout le monde doit soumettre un merge request pour mettre à jour la branche protégée



Workflows de collaboration

Les dépôts distants
Exemple Workflow centralisé
Les branches distantes
Gestionnaire d'intégration
Gitflow
Merge Request dans Gitlab



Introduction

A la différence des SCMs centralisés, la nature distribuée de Git permet beaucoup de flexibilité sur la façon dont les développeurs collaborent

Avec Git, tout développeur peut à la fois contribuer vers les autres dépôts et maintenir un dépôt public sur lequel d'autres vont baser leur travail et auquel ils vont contribuer

=> Il n'y a pas vraiment de règles d'organisation et c'est au choix de l'équipe de mettre en place le *workflow* de collaboration adapté



Les dépôts distants



Dépôts distants

La collaboration sur un projet Git nécessite la gestion de dépôts distants hébergés sur le réseau.

- Il est possible d'en avoir plusieurs avec des droits en lecture ou lecture/écriture différents

La collaboration consiste :

- à récupérer (***pull***)
- ou pousser (***push***) des données vers ses dépôts lorsque l'on doit partager des données

Les opérations de gestion consistent à :

- ajouter/enlever des dépôts
- gérer les branches distantes



git remote et dépôt *origin*

La commande ***git remote*** permet de voir les dépôts configurés

La commande liste les noms courts de chaque dépôt

Si vous avez cloné votre dépôt, vous verrez au moins le dépôt nommé par défaut ***origin*** :

```
$ git clone git://github.com/schacon/ticgit.git
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 193.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```



Option -v

L'option **-v**, affiche l'URL que Git a stockée pour le nom court du dépôt

```
$ git remote -v
origin  git://github.com/schacon/ticgit.git (fetch)
origin  git://github.com/schacon/ticgit.git (push)
```

Dans le cas où il y a plusieurs dépôts

```
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45      git://github.com/cho45/grit.git
defunkt    git://github.com/defunkt/grit.git
koke       git://github.com/koke/grit.git
origin     git@github.com:mojombo/grit.git
```

Dans ce cas, seul le dépôt *origin* a une URL *ssh* permettant l'écriture (*push*)



Ajouter des dépôts

Pour ajouter un nouveau dépôt, il faut utiliser la commande
git remote add [shortname] [url]:

```
$ git remote
```

```
origin
```

```
$ git remote add pb
```

```
git://github.com/paulboone/ticgit.git
```

```
$ git remote -v
```

```
origin  git://github.com/schacon/ticgit.git
```

```
pb      git://github.com/paulboone/ticgit.git
```



Inspecter un référentiel

git remote show [remote-name] permet de visualiser les informations d'un dépôt distant

La commande affiche :

- la liste des URL du référentiel
- La branche qui sera fusionnée localement
- Les branches disponibles sur le dépôt

```
$ git remote show origin
```

```
* remote origin
```

```
URL: git://github.com/schacon/ticgit.git
```

```
Remote branch merged with 'git pull' while on branch master  
master
```

```
Tracked remote branches
```

```
master
```

```
Ticgit
```



Supprimer ou renommer un référentiel

git remote rename permet de renommer un référentiel

```
$ git remote rename pb paul
```

```
$ git remote
```

```
origin
```

```
paul
```

git remote rm permet de supprimer un référentiel

```
$ git remote rm paul
```

```
$ git remote
```

```
origin
```




Commandes de collaboration

Nécessité de synchroniser les branches locales avec le dépôt distant régulièrement :

- **clone** : A l'initialisation, récupère l'ensemble du dépôt et extrait la branche master dans le répertoire de travail
- **fetch** : Se synchronise avec le dépôt (récupération des nouvelles infos) sans modifier le répertoire de travail
- **pull** : Se synchronise avec le dépôt et fusionne les modifications avec le répertoire de travail
- **push** : Pousse ses modifications locales vers le dépôt distant. Opération possible seulement si le dépôt local est à jour



Récupérer un dépôt distant

La commande **git fetch** permet de récupérer un dépôt distant
La commande se connecte au projet distant et récupère toutes les données que l'on ne possède pas déjà

Cette commande ne modifie pas l'espace de travail courant

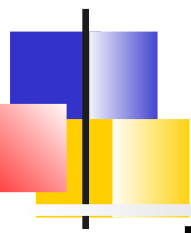
```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch]      master      -> pb/master
* [new branch]      ticgit      -> pb/ticgit
```

=> *La branche master de Paul est accessible localement par pb/master. Il est possible de la fusionner avec une de ses branches ou d'effectuer un check out complet.*



git pull

A la différence de *git fetch*, la commande ***git pull [remote] [branch]*** récupère et fusionne les données automatiquement dans la branche courante. (comme *git clone* qui permet d'initialiser le dépôt et le répertoire de travail)



Pousser vers un dépôt distant

Lorsque votre projet local a atteint un point de développement à partager, il faut utiliser la commande

git push [remote-name] [branch-name]

```
$ git push origin master
```

Cette commande est possible seulement si on a les droits d'écriture sur le dépôt distant et si personne n'a poussé de données entre temps

Si une opération *push* a eu lieu auparavant, il faut d'abord récupérer les données et les fusionner avant de pouvoir les pousser vers le dépôt



Exemple avec un workflow centralisé



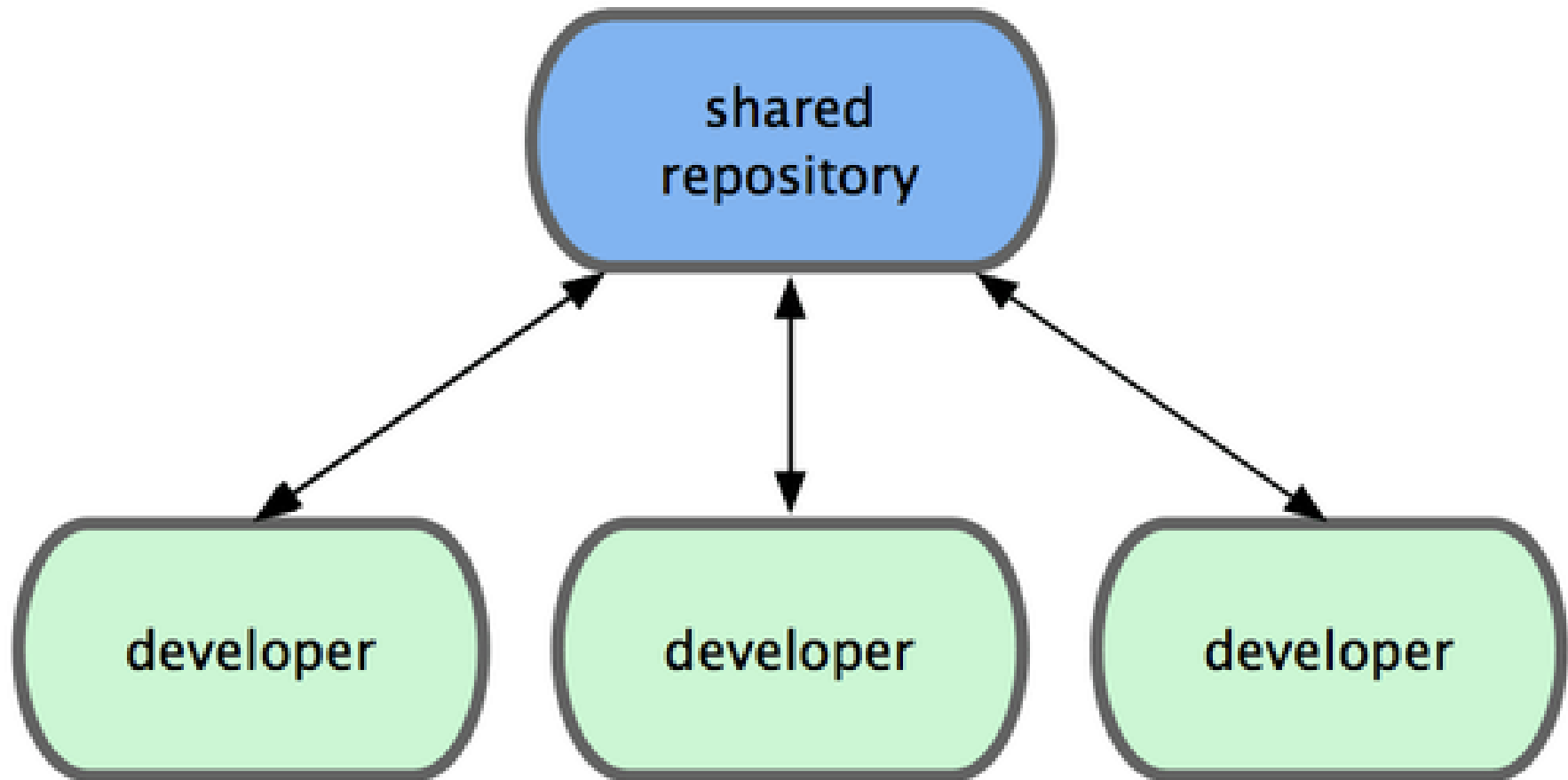
Workflow centralisé

Le **workflow centralisé** consiste à disposer d'un dépôt central avec lequel chaque développeur se synchronise : C'est le modèle des SCMs centralisés

Si 2 développeurs font des changements sur leur copie locale et les commits.

- Le premier développeur committe sans problème
- Le second doit auparavant fusionner le travail du premier avant de committer même si il n'a pas travaillé sur les mêmes fichiers (différence avec *svn*)

Workflow centralisé





Scénario

C 'est le plus simple des workflows.

- Un développeur travaille un temps sur un sujet généralement dans une branche thématique locale
- Lorsque le travail est terminé, il fusionne avec la branche master
- Lorsqu'il veut partager son travail, il récupère la branche master distante et pousse ses modifications.

Scénario





Les branches distantes



Branches distantes

Les **branches distantes** sont des références à l'état des branches sur un référentiel distant.

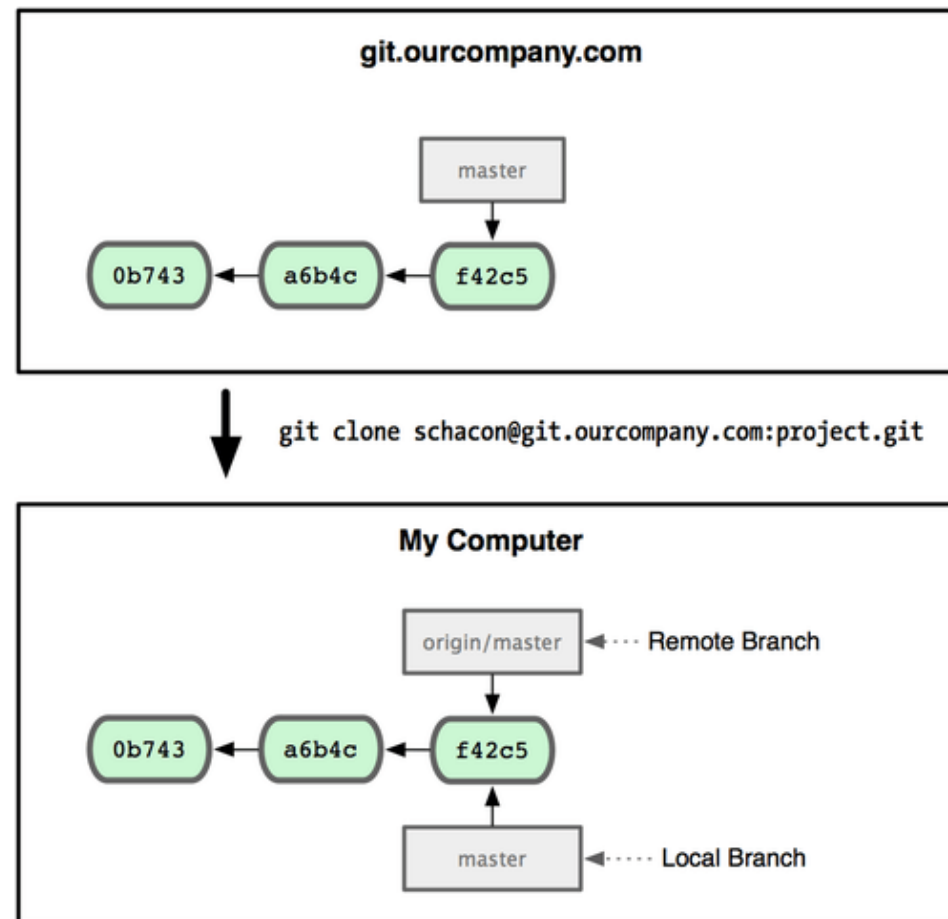
Ce sont des branches locales que l'on ne peut pas modifier.

La référence est mise à jour dès lors qu'il y a une communication réseau

- Les branches distantes sont donc comme des signets qui rappellent l'état de la branche, la dernière fois que l'on s'est connecté au référentiel distant

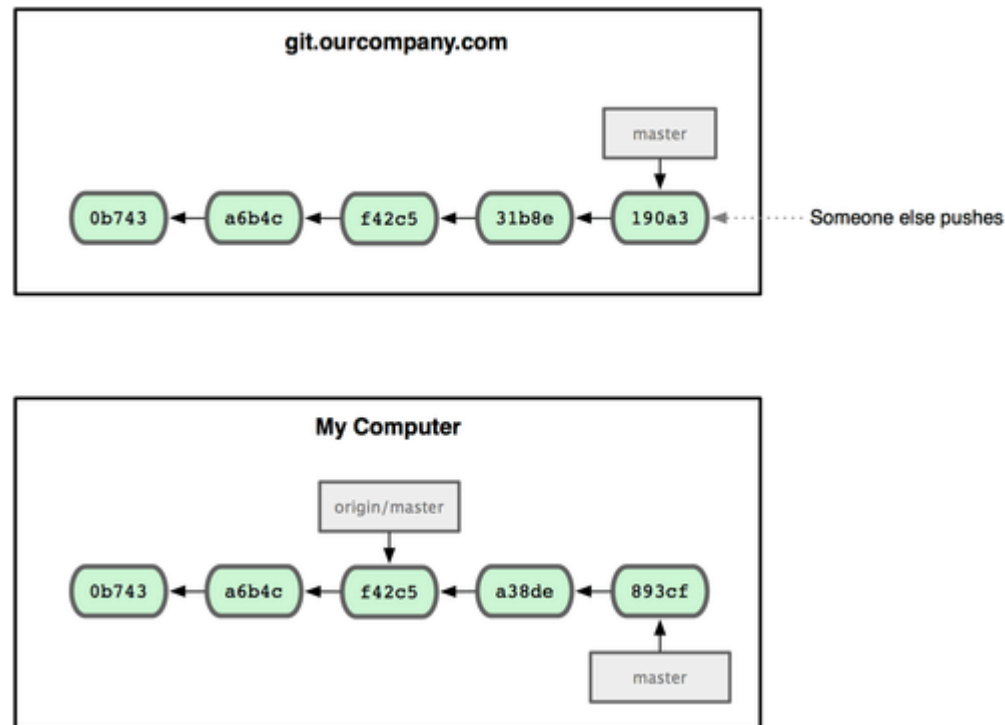
Elles sont référencées dans les commandes *Git* par **(remote)/(branch)**

Exemple après clone



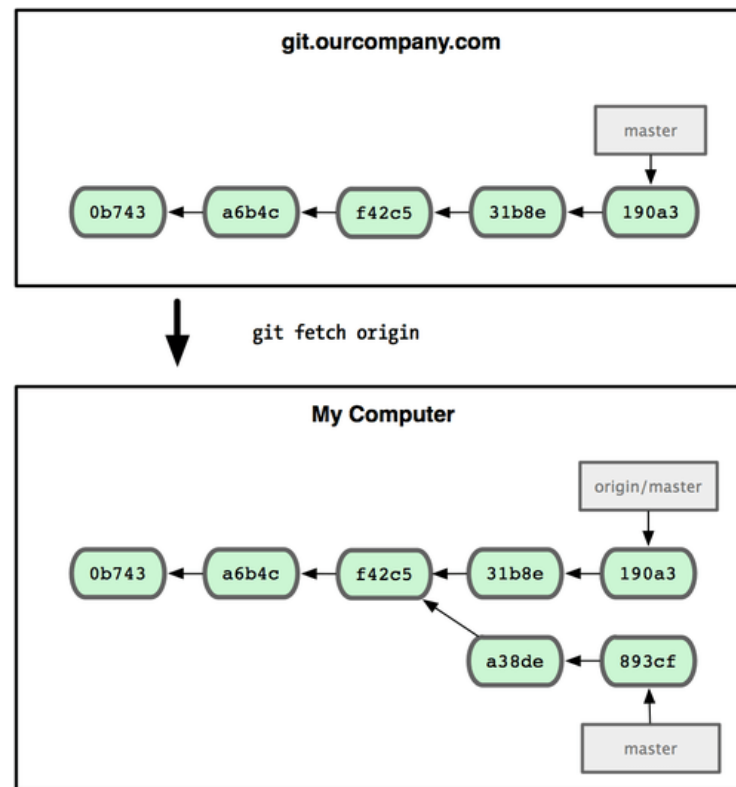
Déplacement

Sans contact avec le serveur d'origine, le pointeur *origin/master* ne se déplace pas



Synchronisation

Pour synchroniser une branche distante, on exécute la commande ***git fetch origin*** qui rapatrie les nouvelles données et met à jour la base de données locale en déplaçant le pointeur *origin/master* à sa nouvelle position





Checkout et fusion

Lorsque l'on utilise la commande *fetch*, le répertoire de travail n'est pas modifié

Pour fusionner la branche distante avec la branche actuelle de travail, on peut utiliser :

```
$ git merge remote/branch
```

Si on souhaite créer une propre branche basée sur le pointeur distant :

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

```
Branch correctionserveur set up to track remote branch refs/  
remotes/origin/correctionserveur.
```

```
Switched to a new branch "correctionserveur"
```



Branche de suivi

L'extraction d'une branche locale à partir d'une branche distante crée automatiquement une **branche de suivi**.

Les branches de suivi sont des branches locales qui sont en relation directe avec une branche distante

Dans une branche de suivi *git push*, et *git pull* sélectionne automatiquement le serveur impliqué

C'est le même mécanisme lorsque l'on clone un dépôt



Branches de suivi

Il y a plusieurs façons de créer des branches de suivi :

L'option *--track* :

```
$ git checkout --track origin/serverfix
```

Si la branche n'existe pas localement et que son nom correspond exactement à une branche de suivi, on peut utiliser le raccourci :

```
$ git checkout serverfix
```

Si l'on veut renommer la branche

```
$ git checkout -b sf origin/serverfix
```

Enfin, si on veut utiliser une branche locale existante :

```
$ git branch --set-upstream-to origin/serverfix
```



Push

Lorsque l'on veut partager une branche, il faut la pousser explicitement dans un référentiel distant

L'option **-u** permet de configurer la branche locale comme branche de suivi

```
$ git push -u origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new branch]      serverfix -> serverfix
```

=> Avec Git, on peut utiliser des branches privées et ne pousser que les branches sur lesquelles on souhaite collaborer.



Syntaxe complète

Lorsqu'on précise la branche dans la commande *push*, on utilise en fait un raccourci. La syntaxe complète est plutôt :

```
$ git push origin serverfix:serverfix
```

Ce qui veut dire

« Recopier ma branche locale nommée **serverfix** dans la branche distante nommée **serverfix** »

Si l'on veut donner un autre nom à la branche distante, on peut utiliser :

```
$ git push origin serverfix:autrenom
```



Effacer une branche distante

Effacer une branche distante consiste à pousser un contenu vide vers la branche du serveur

```
$ git push origin :correctionserveur  
To git@github.com:schacon/simplegit.git  
- [deleted]                correctionserveur
```



Git prune

La suppression d'une branche sur le serveur ne supprime pas les branches distantes (présente en locale)

Pour supprimer les branches distantes pointant sur des branches n'existant plus

```
$ git remote prune origin
```



Partager les tags

Par défaut la commande *git push* ne transfère pas les tags vers le référentiel distant, il faut explicitement les pousser après leur création

```
$ git push origin v1.5
```

```
Counting objects: 50, done.
```

```
Compressing objects: 100% (38/38), done.
```

```
Writing objects: 100% (44/44), 4.56 KiB, done.
```

```
Total 44 (delta 18), reused 8 (delta 1)
```

```
To git@github.com:schacon/simplegit.git
```

```
* [new tag]          v1.5 -> v1.5
```

=> Désormais, lorsque quelqu'un clone ou récupère les données du référentiel, il récupère également les tags.



Patterns de workflow utilisant les branches



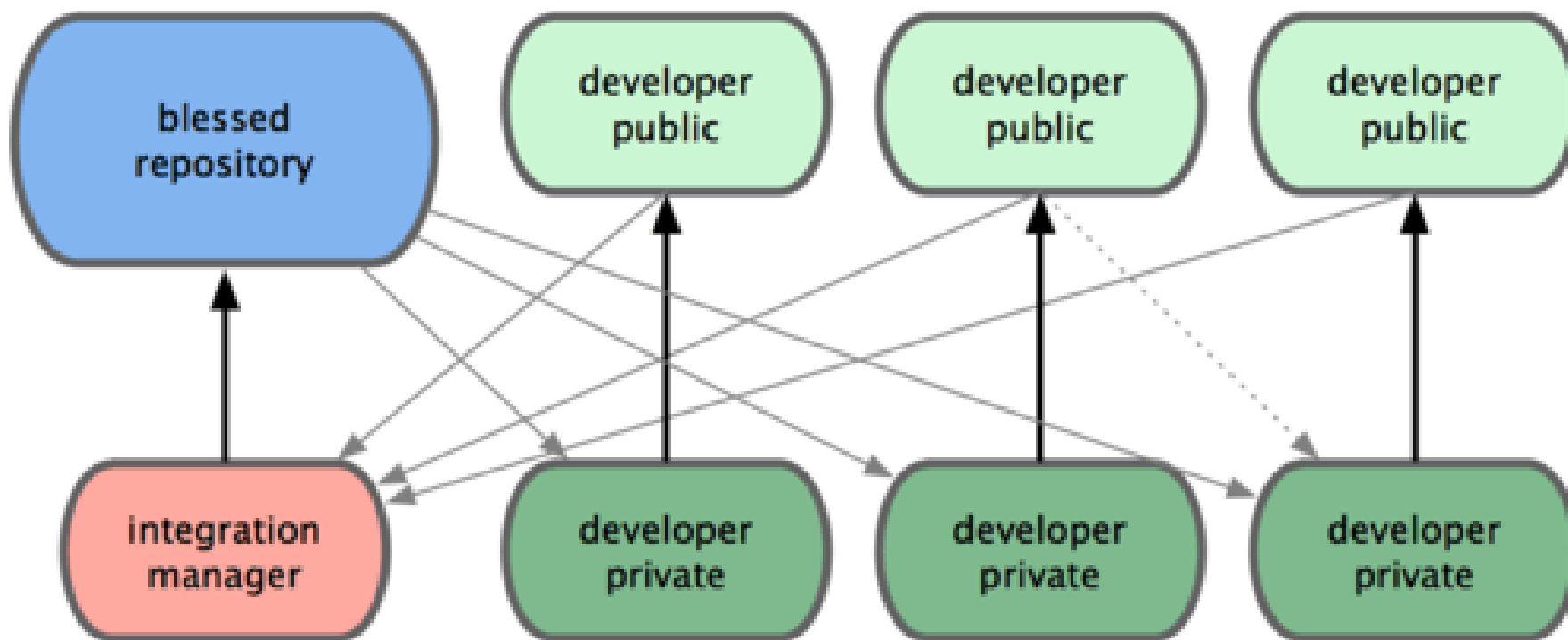
Introduction

Les SCMs distribués ont introduit différents workflows de collaboration entre développeurs :

- Projets OpenSource (Linux, Github, ...) :
Workflow avec intégrateur basé sur les *pull-request*
- Editeur logiciel avec maintenance concurrente de plusieurs releases : **Gitflow**
- Projet DevOps avec déploiement continu :
GitlabFlow basé sur les merge-request



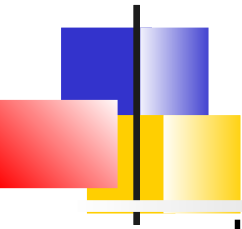
Gestionnaire d'intégration





Étapes

1. L'intégrateur pousse vers son dépôt public.
2. Un contributeur clone ce dépôt et introduit des modifications.
3. Le contributeur pousse son travail sur son dépôt public.
4. Le contributeur envoie à l'intégrateur un e-mail de demande pour tirer depuis son dépôt. (***pull-request***)
5. Le mainteneur ajoute le dépôt du contributeur comme dépôt distant et fusionne localement.
6. Si cela lui paraît adéquat, le mainteneur pousse les modifications fusionnées sur le dépôt principal



Gitflow

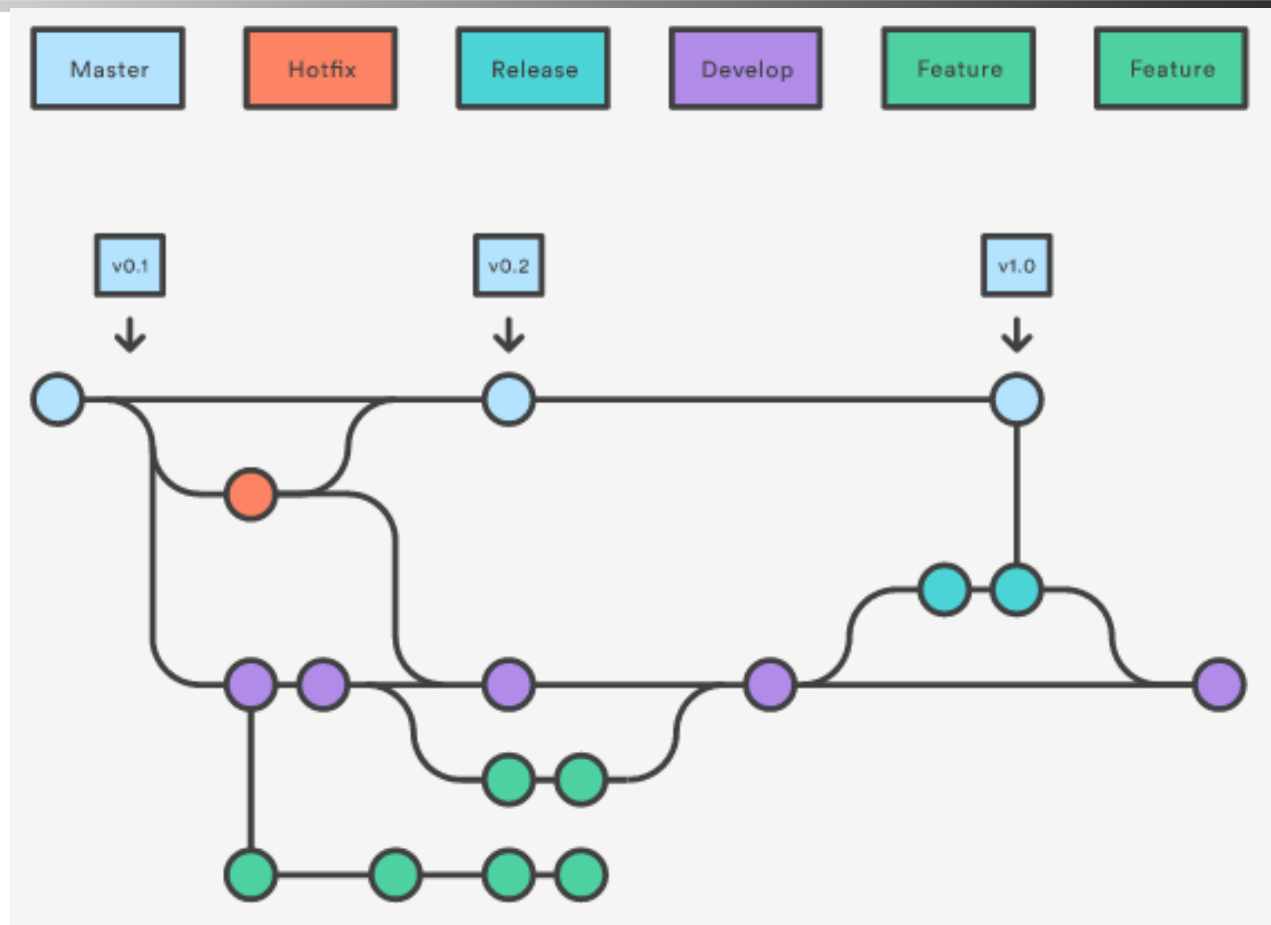
Le workflow **Gitflow** définit un modèle de branches orientées vers la production de releases maintenables

- Adapté pour projets d'édition logiciel
- Des rôles très spécifiques sont assignés aux différentes branches et Gitflow définit quand et comment elles doivent interagir

Il utilise des :

- Des branches **longues** (master, dev) qui existent pendant tout le projet
- Des branches **courtes** qui sont supprimées dès lors qu'elles ont atteint leur but

Branches Gitflow





Déclinaisons

On peut décliner *Gitflow* avec d'autres branches annexes comme des branches de revue de code

- Elles permettent de valider des modifications avant de les intégrer dans une branche supérieure.
Exemple de *Gerrit*

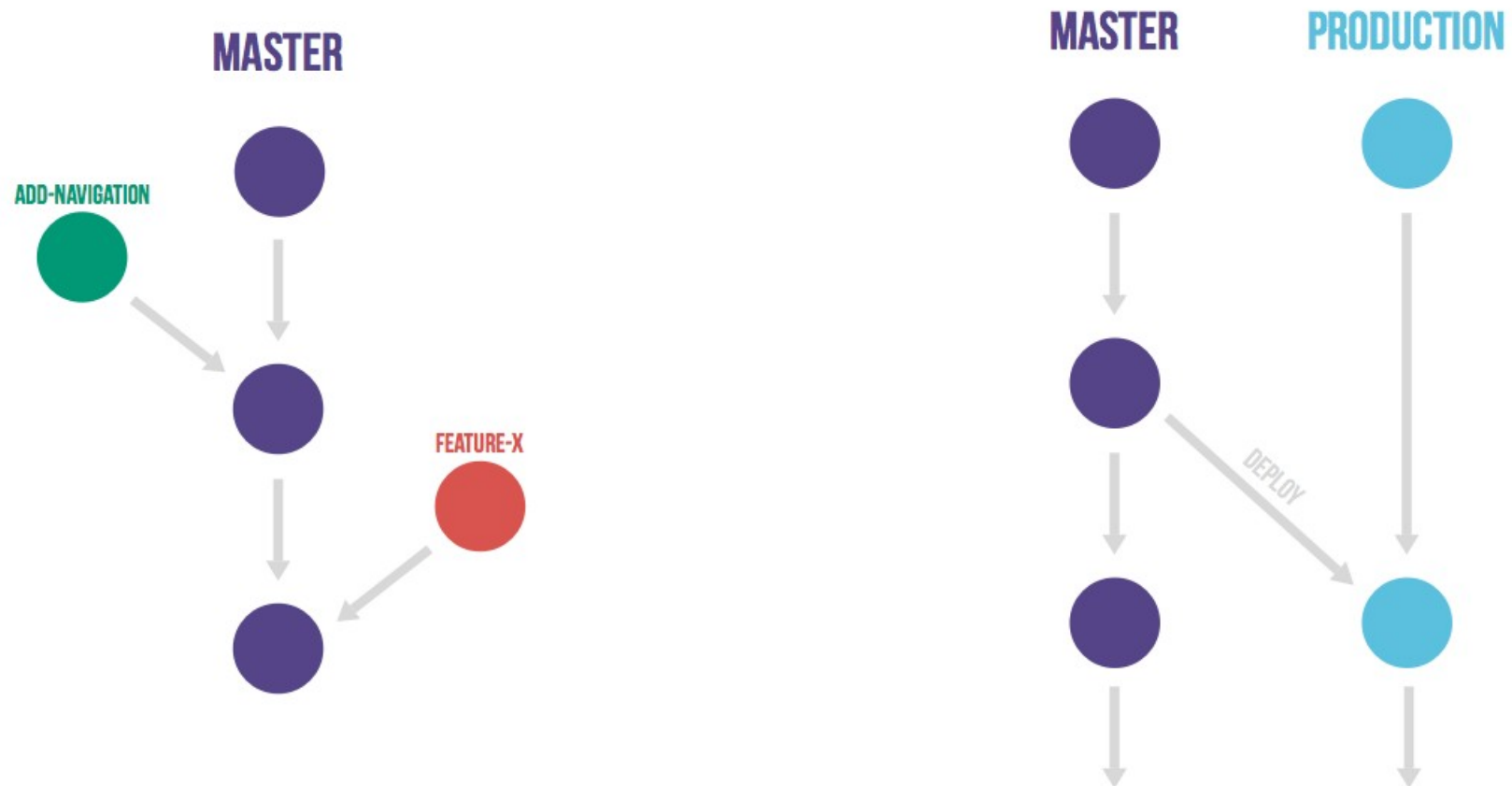


Gitlab Flow

Gitlab Flow est une stratégie simplifiée d'utilisation des branches pour un développement piloté par les features ou le suivi d'issues

- 1) Les fix ou fonctionnalités sont développés dans une feature branch
- 2) Via un merge request, elles sont intégrées dans la branche master
- 3) Il est possible d'utiliser d'autres branches :
 - *production* : Chaque merge est taggée et correspond à une livraison dans l'environnement de production
 - *release* : Branche de release. Chaque merge est taggée et correspond à une distribution de release. Les Bug fixes sont repris de master via des cherry-picks dans les branches de release impactées

Features, Master and Production





Workflow typique

- 1) Les travaux sont effectués localement dans une branche
- 2) Ils sont ensuite poussés sur Gitlab
- 3) Un merge request est créé
- 4) *Gitlab* permet alors d'effectuer une revue de code ainsi que de collaborer sur les modifications en cours
- 5) Éventuellement, ces modifications peuvent être déployées sur une « Review Apps »
- 6) Des approbations peuvent être demandées au *maintainers* avant le merge dans la branche master



Déploiements

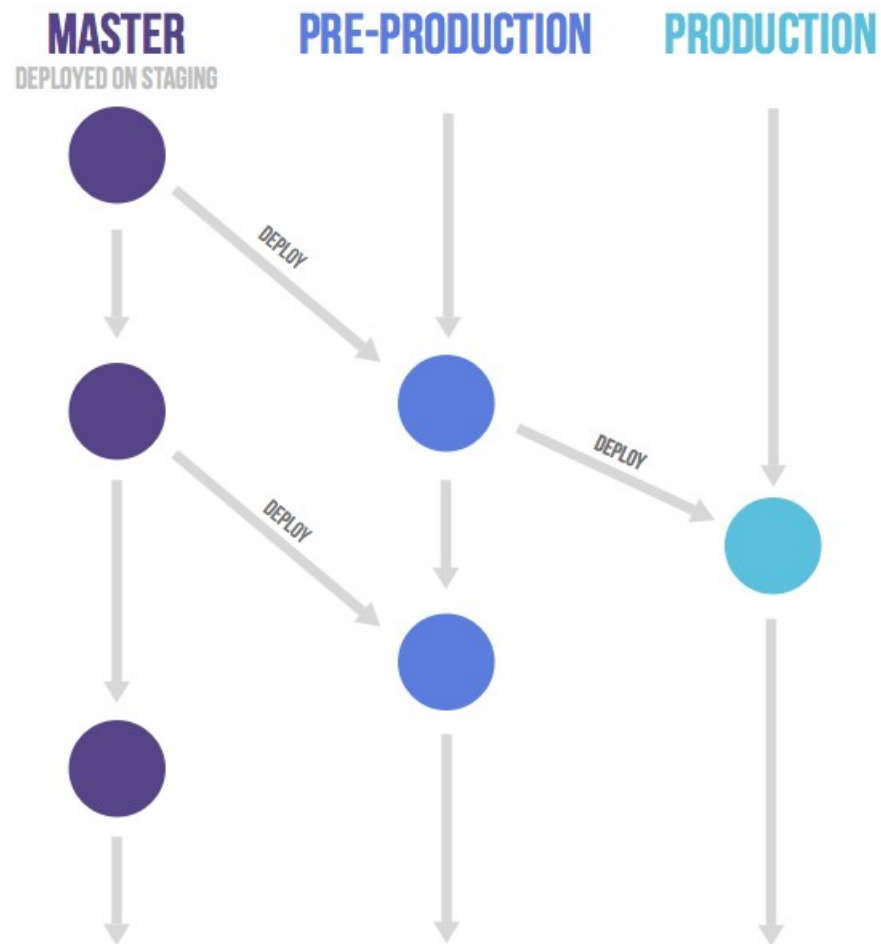
Les déploiements s'effectuent à partir de la branche *master*

Gitlab est capable de gérer plusieurs **environnements** de déploiement

- Pour chaque environnement, une branche est créé
=> L'historique des déploiements sur un environnement particulier est aisément consultable



Branches d'environnement





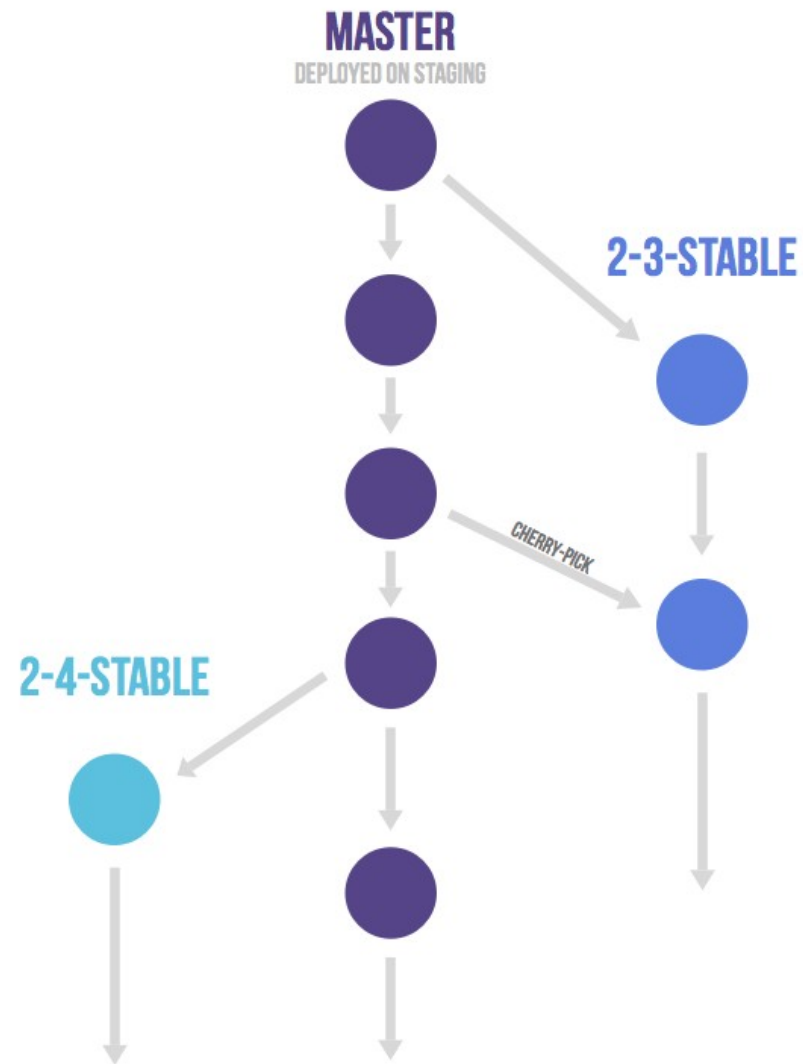
Releases

Pour la distribution de software, il est possible de mettre en place des **branches de release**

- Lors de la préparation d'une release, une branche stable est créé à partir du *master*
- Un tag est créé pour chaque version
- Les bugs critiques trouvés à posteriori sont appliqués via des Cherry-pick
(Ne pas committer directement dans la branche stable)

Gitlab permet de visualiser les *Releases* d'un projet via l'UI et de fournir les artefacts construits via téléchargement

Branches de releases





Merge Request dans Gitlab



Introduction (1)

Le ***Merge Request*** est la base de la collaboration sur Gitlab

Un MR permet :

- Comparer les changements entre 2 branches
- Revoir et discuter des modifications de code
- Voir l'appli. en fonctionnement (*Review Apps*)
- Exécuter une pipeline
- Empêcher une fusion trop précoce avec le *WIP*
- Visualiser le processus de déploiement
- ...



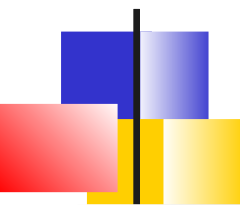
Introduction (2)

- Supprimer automatiquement la branche associée à la modification
- Assigner la MR à un responsable
- Affecter un milestone
- Utiliser des workflows de collaboration via les labels
- Faire un suivi du temps
- Résoudre les conflits de merge via l'UI
- Autoriser les fast-forward
- ...




Cycle de vie d'un MR

- 1) Au démarrage d'un nouveau travail, le développeur crée un *merge request*.
Le travail n'est pas prêt à être fusionné mais la collaboration et la revue de code peuvent commencer dans une feature branch.
Le *Merge Request* est préfixé par **WIP**
- 2) Lorsque la fonctionnalité est prête, le développeur assigne le *merge request* à un des membres du projet (un mainteneur en général)
- 3) Le mainteneur a le choix entre effectuer la fusion dans master, demander au développeur des améliorations, abandonner la MR
- 4) Lorsque la feature branch est fusionnée, elle est détruite .



Vue projet

 GitLab Community Edition

Overview

Repository

Issues8,730



Merge Requests472

CI / CD

Wiki

Snippets


Settings

GitLab /  GitLab.org /  GitLab Community Edition

Merge Requests

Edit Merge RequestsNew merge request

Open472Merged11,188Closed1,969All13,629

 Search or filter results...

Last created

test MR

!13679 · opened 17 minutes ago by Mike Greiling

updated 14 minutes ago

Add docs for group issues page and group merge requests page

!13678 · opened 20 minutes ago by Victor Wu

updated 2 minutes ago

Docs update links guideline to inline links

!13677 · opened 36 minutes ago by Marcia Ramos 10.0 Documentation docs-update

updated 34 minutes ago

WIP: Clean up new dropdown styles 0 of 1 task completed

!13676 · opened 50 minutes ago by Winnie Hellmann 10.0 Deliverable UI polish frontend

updated 15 minutes ago

Greatly reduce test duration for git_access_spec

!13675 · opened 58 minutes ago by Robert Speicher 10.0 Edge backstage performance technical debt test

updated 20 minutes ago

Implement new system note icons 0 of 11 tasks completed

!13673 · opened about 3 hours ago by Bryce Johnson 10.0 Deliverable frontend

updated less than a minute ago

WIP: Prepare 9.5 RC6

!13672 · opened about 3 hours ago by Jose Ivan Vargas Lopez 9-5-stable Release

updated about an hour ago

Use Gitaly 0.33.0 0 of 11 tasks completed

!13671 · opened about 4 hours ago by Jacob Vosmaer (GitLab) 9.5 Gitaly Pick into Stable

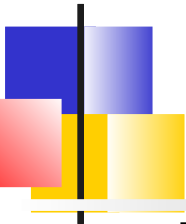
updated about 4 hours ago

[WIP] Make the import take subgroups into account 0 of 9 tasks completed

!13670 · opened about 5 hours ago by Bob Van Landuyt 10.0 Platform

updated about 5 hours ago

217



Commentaires et discussions

Des **commentaires** peuvent être associés aux différents objets de Gitlab dont les MR

Un commentaire peut être transformé en **discussion**. (via l'UI ou via un email)

Le statut de résolution d'une discussion peut affecter la merge request

- La discussion démarre avec un statut *unresolved*
- Elle s'applique en général à un *diff* d'un des commits du merge request



Commentaires et discussions

Pour créer une discussion sur un commit

- Afficher les commits liés au MR
- Sur un commit, accéder à l'onglet *Changes* et laisser un commentaire
- La discussion apparaît dans l'onglet discussions du MR et peut être résolue via le bouton « *Resolve Discussion* »

Il est possible de

- voir toutes les discussions non résolues
- De déplacer les discussions non résolues vers une issue



Conflits

Lorsqu'une MR a des conflits, il est possible de les résoudre via l'UI

GitLab résout les conflits en créant un commit de merge dans la branche source.

Le commit peut alors être testé avant d'affecter la branche cible.



Squash

Lors d'un merge, il est possible de convertir tous les commits du merge en un seul et donc d'avoir un historique plus clair : ***squash***

Le message de commit est alors :

- Repris du premier message de commit multi-lignes
- Le titre du merge request si il n'y a pas de messages multi-lignes

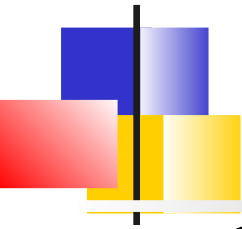
Il peut être personnalisé au moment du merge



Méthodes de merge

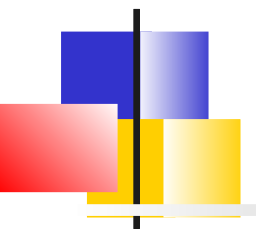
Les méthodes de merge ont une influence sur l'historique du projet :

- **Merge commit (défaut)** : Chaque fusion crée un commit de merge
- **Merge commit avec historique semi-linéaire** :
Chaque fusion crée un commit de merge mais la fusion n'est possible que si c'est une fast-forward
Si un conflit arrive, l'utilisateur a la possibilité de rebaser
- **Fast-forward merge** : Pas de commit de merge, seules les fast-forward sont possibles
Si un conflit arrive, l'utilisateur a la possibilité de rebaser



Vérifications avant merge

- 2 vérifications effectuées avant un merge peuvent être configurées :
- Vérifier que la pipeline réussisse
 - Vérifier que les discussions sont closes



Configuration Merge Request

Merge requests

Collapse

Choose your merge method, options, checks, and set up a default merge request description template.

Merge method

This will dictate the commit history when you merge a merge request

- ☒ Merge commit
Every merge creates a merge commit
- ☐ Merge commit with semi-linear history
Every merge creates a merge commit
Fast-forward merges only
When conflicts arise the user is given the option to rebase
- ☐ Fast-forward merge
No merge commits are created
Fast-forward merges only
When conflicts arise the user is given the option to rebase

Merge options

Additional merge request capabilities that influence how and when merges will be performed

- ☐ Automatically resolve merge request diff discussions when they become outdated
- ☒ Show link to create/view merge request when pushing from the command line

Merge checks

These checks must pass before merge requests can be merged

- ☐ Pipelines must succeed
Pipelines need to be configured to enable this feature. [?](#)
- ☐ All discussions must be resolved



Approbateurs

Si l'installation Gitlab a été configuré il est possible de configurer la politique d'approbation d'une MR. Les approbateurs des *MRs* sont configurés au niveau du projet.

- => On peut alors indiquer des membres du projet (ou du groupe ou du groupe partagé) ainsi qu'un nombre



Écran de configuration

Merge request approvals

Set a number of approvals required, the approvers and other approval settings. [Learn more about approvals.](#)

Add approvers

Members

No. approvals required

All members with Developer role or higher and code owners (if any)

0

Edit

Add approvers

- ☐ Require approval from code owners ?
- ☒ Can override approvers and approvals required per merge request ?
- ☒ Remove all approvals in a merge request when new commits are pushed to its source branch
- ☒ Prevent approval of merge requests by merge request author ?
- ☐ Prevent approval of merge requests by merge request committers ?
- ☐ Require user password to approve ?

Save changes



Pour aller plus loin

Refs

Quelques outils utiles
Hooks et personnalisation



Refs



Introduction

Une ***ref*** est une méthode indirecte pour référencer un commit (~ un alias vers un hash de commit)

C'est le mécanisme interne de Git pour représenter les branches et les tags

Elles sont stockées sous forme de fichiers dans le répertoire ***.git/refs***



Contenu de *.git/refs*

Le répertoire contient :

- Le sous-répertoire ***heads*** qui contient un fichier texte par branche locale, le contenu contient la clé de hash de la branche
- Le sous-répertoire ***tags*** contient un fichier par tag. Chaque fichier contient la clé de hash associé au tag
- Le sous-répertoire ***remotes*** contient un sous-répertoire par dépôt distant et dans chaque sous-répertoire les branches distantes



Spécifier une ref

Lors de l'utilisation d'une commande git, on peut passer le nom court d'une *ref* :

```
git show somefeature
```

Ou son nom long

```
git show refs/branch/somefeature
```

Lors de l'utilisation du nom court, Git résout le nom court en un nom long. Il réussit s'il n'y a pas d'ambiguïté (pas une branche qui a le même nom d'un tag)



Refs spéciales

En plus des refs stockées dans le répertoire *refs*, il existe des refs directement à la racine de *.git*

- **HEAD** : La branche ou commit courant de l'espace de travail.
- **FETCH_HEAD** : Les branches distantes le plus récemment récupérée.
- **ORIG_HEAD** : Un backup de HEAD avant que l'on effectue des changements dessus.
- **MERGE_HEAD** : Le ou les commit(s) que l'on est train de fusionner dans la branche courante avec *git merge*.
- **CHERRY_PICK_HEAD** : Le commit que l'on utilise avec avec *git cherry-pick*.

HEAD est la seule *ref* que l'on utilise de façon courante



Contenu des refs spéciales

Le contenu des *refs spéciales* dépend de leur type et de l'état du dépôt local

Par exemple, HEAD peut contenir soit :

- Une ref symbolic : référence vers une autre ref => HEAD est synchronisé avec une branche
- Ou un hash de commit => HEAD est dans un état détaché



Raccourcis reflog

Git maintient un historique des références où sont passés les pointeurs HEAD et ceux des branches sur les derniers mois : le reflog.

```
$ git reflog
```

```
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
```

```
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
```

```
1c002dd HEAD@{2}: commit: added some blame and merge stuff
```

```
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
```

```
95df984 HEAD@{4}: commit: # This is a combination of two commits.
```

```
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
```

```
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

On peut spécifier des anciens commits avec ces données et la notation @(n)

```
$ git show HEAD@{5}
```

Ou en indiquant une date relative

```
$ git show master@{yesterday}
```

```
$git show HEAD@{2.months.ago}
```



Exemple

```
400e4b7 HEAD@{0}: checkout: moving from master to HEAD~2
0e25143 HEAD@{1}: commit (amend): Integrate some awesome feature into `master`
00f5425 HEAD@{2}: commit (merge): Merge branch ';feature';
ad8621a HEAD@{3}: commit: Finish the feature
```

Traduction :

- On a effectué une extraction de HEAD~2
- Avant, on a amendé un message de commit
- Avant, on a fusionné la branche feature dans master
- Avant on a commité un instantané



Quelques commandes utiles



Alias

```
git config global alias.co checkout  
git config global alias.br branch  
git config global alias.unstage 'reset HEAD -'  
git config global alias.visual '!gitk'
```



Sélection de révision

On peut faire référence à un commit par sa clé de hachage SHA-1 mais d'autres façons existent.

On peut utiliser une version courte de la clé de hachage : au minimum 4 caractères nom ambigu

Par exemple, ces commandes sont généralement équivalentes :

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
```

```
$ git show 1c002dd4b536e7479f
```

```
$ git show 1c002d
```

L'option **--abbrev-commit** de *git log* permet de voir les SHA courts adéquats

```
$ git log --abbrev-commit --pretty=oneline
```

```
ca82a6d changed the version number
```

```
085bb3b removed unnecessary test code
```

```
a11bef0 first commit
```



Référence de branche

La façon la plus directe de référencer un commit nécessite qu'une branche pointe dessus

```
$ git show topic1
```

Si l'on veut accéder à la clé de hachage correspondante on peut utiliser ***rev-parse***

```
$ git rev-parse topic1  
ca82a6dff817ec66f44342007202690a93763949
```



Référence d'ascendance

Si on place le caractère ^ à la fin d'une référence, cela permet d'accéder à son parent

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800
    Merge commit 'phedders/rdocs'
```

Dans le cas d'une fusion où le commit à plusieurs parents, on peut indiquer les autres parents en ajoutant en indice :

```
$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date: Wed Dec 10 22:22:03 2008 +0000
    Some rdoc changes
```




Référence d'ascendance

Le caractère \sim peut également être utilisé.

Ex : $HEAD \sim$, signifie le parent de $HEAD$ ($\Leftrightarrow HEAD^{\wedge}$)

Ce caractère peut être utilisé pour accéder au grand-parent

Ex : $HEAD \sim 2 \Leftrightarrow HEAD^{\wedge \wedge}$

Les syntaxes peuvent être combinées

Ex : $HEAD \sim 3^2$



Intervalles de commit

Il est possible de spécifier des intervalles de commit.

Cela peut être utile par exemple pour répondre à la question : « *Quels travaux dans cette branche n'ont pas encore été fusionnés dans la branche principale ?* »

La syntaxe la plus courante utilise la notation ..

Cela demande à Git de trouver tous les commits accessibles de la première branche mais pas accessibles de la seconde

Exemple

```
$ git log master..experiment
```

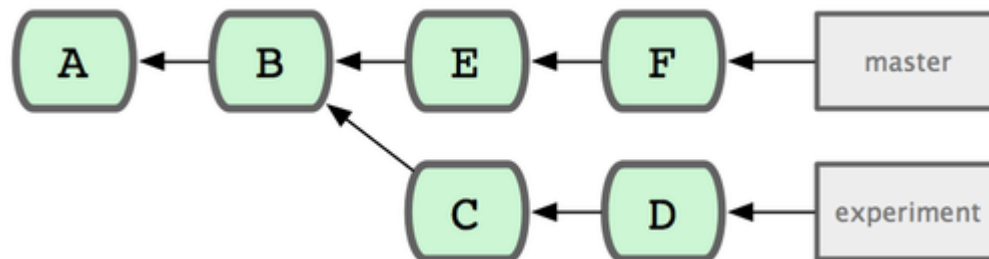
D

C

```
$ git log experiment..master
```

F

E



Voir ce qui va être poussé sur le référentiel distant :

```
$ git log origin/master..HEAD
```

```
$ git log origin/master..
```



Plusieurs branches

Il est possible de spécifier plus de 2 branches en utilisant les caractères `^` ou **`--not`** devant chaque référence à partir desquels les commits ne sont pas accessibles.

Les 3 commandes suivantes sont équivalentes :

```
$ git log refA..refB
```

```
$ git log ^refA refB
```

```
$ git log refB --not refA
```

Appliqué à plus de 2 branches :

```
$ git log refA refB ^refC
```

```
$ git log refA refB --not refC
```



Triple points

La syntaxe **...** permet de spécifier les commits accessibles d'une des deux branches mais pas par les 2

Associé à l'option **--left-right**, elle permet de visualiser de quelle branche proviennent les commits

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```



cherry-pick

cherry-pick permet d'appliquer les changements effectués par un ou plusieurs commit

```
git cherrypick <commit>...
```

Commande très appréciée des intégrateurs

Elle peut provoquer des conflits qui sont alors résolus comme dans un rebase :

```
git cherrypick --continue ou --abort
```



Indexation interactive

Si l'on utilise l'option **-i** ou **--interactive** avec la commande *add*, Git propose un mode interactif

```
$ git add -i
```

	staged	unstaged	path
1:	unchanged	+0/-1	TODO
2:	unchanged	+1/-1	index.html
3:	unchanged	+5/-1	lib/simplegit.rb

```
*** Commands ***
```

1: status	2: update	3: revert	4: add untracked
5: patch	6: diff	7: quit	8: help

```
What now>
```

Ensuite, on choisit une commande puis les numéros de fichiers sur lesquels on veut appliquer la commande



Indexation interactive

Si l'on utilise l'option **-i** ou **--interactive** avec la commande *add*, Git propose un mode interactif

```
$ git add -i
```

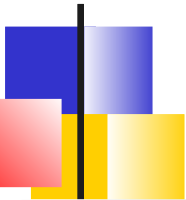
	staged	unstaged	path
1:	unchanged	+0/-1	TODO
2:	unchanged	+1/-1	index.html
3:	unchanged	+5/-1	lib/simplegit.rb

```
*** Commands ***
```

1: status	2: update	3: revert	4: add untracked
5: patch	6: diff	7: quit	8: help

```
What now>
```

Ensuite, on choisit une commande puis les numéros de fichiers sur lesquels on veut appliquer la commande



Indexation partielle de fichier

La commande **patch** en mode interactif permet d'indexer un sous-ensemble de modifications d'un même fichier

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
```

```
index dd5ecc4..57399e0 100644
```

```
--- a/lib/simplegit.rb
```

```
+++ b/lib/simplegit.rb
```

```
@@ -22,7 +22,7 @@ class SimpleGit
  end
```

```
  def log(treeish = 'master')
```

```
  -   command("git log -n 25 #{treeish}")
```

```
  +   command("git log -n 30 #{treeish}")
```

```
  end
```

```
  def blame(path)
```

```
  Stage this hunk [y,n,a,d,/,j,J,g,e,]?
```



Options possibles

Les options possibles pour chaque modification sont alors :

- y - Mettre en staging cette partie
- n - ne pas mettre en staging cette partie
- a - Mettre en staging cette partie et toutes celles restantes dans ce fichier
- d - ne pas mettre en staging cette partie ni aucune de celles restantes dans ce fichier
- g - sélectionner une partie à voir
- / - chercher une partie correspondant à la regexp donnée
- j - laisser cette partie non décidée, voir la prochaine partie non encore décidée
- J - laisser cette partie non décidée, voir la prochaine partie
- k - laisser cette partie non décidée, voir la partie non encore décidée précédente
- K - laisser cette partie non décidée, voir la partie précédente
- s - couper la partie courante en parties plus petites
- e - modifier manuellement la partie courante
- ? - afficher l'aide



Mise de côté

La commande ***git stash*** permet de mettre côté un travail en cours sans le committer

On peut alors basculer sur une autre branche et revenir plus à l'état du répertoire de travail mis de côté

```
$ git stash
```

```
Saved working directory and index state \
```

```
"WIP on master: 049d078 added the index file"
```

```
HEAD is now at 049d078 added the index file
```

```
(To restore them type "git stash apply")
```

Le répertoire de travail est alors propre :

```
$ git status
```

```
# On branch master
```

```
nothing to commit, working directory clean
```



git stash

L'option ***list*** permet de voir les états mis de côté

```
$ 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 récupérer un élément de la pile utiliser ***git stash apply*** ou ***git stash apply stash@(n)*** :

```
$ git stash apply
```

```
# On branch master
```

```
# Changes not staged for commit:
```

```
#   (use "git add <file>..." to update what will be committed)
```

```
#
```

```
#       modified:   index.html
```

```
#       modified:   lib/simplegit.rb
```

```
#
```

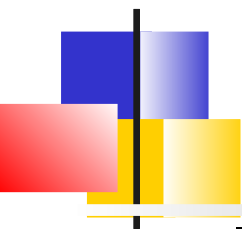
Il n'est pas nécessaire de réappliquer un stash sur la branche d'origine, dans ce cas Git tente de fusionner



Autres commandes de *git stash*

Les autres commandes sont :

- ***git stash drop*** : Supprime un stash
- ***git stash pop*** : Applique le stash et le supprime
- ***git stash branch <branch_name>*** :
Création d'une branche à partir d'un stash



Réécriture de l'historique

Il est possible de modifier des commits antérieurs :

- Changer les messages
- Modifier les fichiers du commit
- Changer l'ordre des différents commits
- Fusionner ou diviser des commits
- Supprimer des commits

Naturellement, tout cela doit être fait avant de partager ses modifications avec les autres développeurs



Modification du dernier commit

Pour changer le dernier commit :

```
$ git commit --amend
```

Pour modifier un commit plus ancien, on peut utiliser la commande **rebase** qui travaille sur une série de commits allant jusqu'au HEAD

En mode interactif, cette commande permet de s'arrêter après chaque commit que l'on veut modifier (changer les messages, ajouter des fichiers ou autre).

La commande *rebase* prend en paramètre le premier commit sur lequel on veut travailler

Ensuite, pour chaque commit jusqu'au HEAD, on peut spécifier les actions que l'on veut effectuer en éditant un fichier

A la sauvegarde du fichier, il est possible de modifier le commit courant par **git commit --amend**

Puis de continuer, l'opération de rebase via **git rebase --continue**



Example

```
$ git rebase -i HEAD~3
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# 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
```




Exemple

Changement du message du 3ème dernier commit

```
edit f7f3f6d changed my name a bit
```

```
pick 310154e updated README formatting and added blame
```

```
pick a5f4a0d added cat-file
```

Sauvegarde et sortie de l'éditeur

```
$ git rebase -i HEAD~3
```

```
Stopped at 7482e0d... updated the gemspec to hopefully work better
```

```
You can amend the commit now, with
```

```
git commit -amend
```

Once you're satisfied with your changes, run

```
git rebase -continue
```

Commit du nouveau message

```
$ git commit -amend
```

Continuer le process de rebasing (dans ce cas les 2 commits suivants sont utilisés tel quel)

```
$ git rebase --continue
```



Réordonner les commits

Par exemple pour supprimer un commit et réordonner les 2 restants

Le fichier de commande passe de

```
pick f7f3f6d changed my name a bit  
pick 310154e updated README formatting and added blame  
pick a5f4a0d added cat-file
```

à

```
pick 310154e updated README formatting and added blame  
pick f7f3f6d changed my name a bit
```



Fusionner des commits

Pour transformer les 3 commits en un commit unique, le fichier est alors :

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

A la sauvegarde, Git revient sur l'éditeur pour fusionner les messages

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit
```

```
# This is the 2nd commit message:
```

```
updated README formatting and added blame
```

```
# This is the 3rd commit message:
```

```
added cat-file
```



Diviser un commit

Si l'on veut séparer un commit en plusieurs, il faut annuler le commit puis mettre partiellement en zone de staging les fichiers modifiés, les committer et répéter ces opérations autant de fois que l'on veut

Par exemple, si l'on veut diviser le second commit

```
pick f7f3f6d changed my name a bit
```

```
edit 310154e updated README formatting and added blame
```

```
pick a5f4a0d added cat-file
```

A la sauvegarde de l'éditeur, Git retourne au parent du premier commit de la liste, applique le premier commit (f7f3f6d), applique le second (310154e), et vous donne accès à la console

Il est possible alors d'annuler ce commit et de placer les modifications en zone de staging en plusieurs étapes



Diviser un commit (2)

```
$ git reset HEAD^  
$ git add README  
$ git commit -m 'updated README formatting'  
$ git add lib/simplegit.rb  
$ git commit -m 'added blame'  
$ git rebase -continue
```

Git applique ensuite le dernier commit

```
$ git log -4 --pretty=format:"%h %s"  
1c002dd added cat-file  
9b29157 added blame  
35cfb2b updated README formatting  
f3cc40e changed my name a bit
```



filter-branch

Pour changer un grand nombre de commit (changer un email, supprimer un fichier de tous les commits), il est possible d'utiliser la commande ***filter-branch***

Supprimer un fichier de tous les commits :

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
```

```
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)  
Ref 'refs/heads/master' was rewritten
```

Désigner un sous-répertoire comme la nouvelle racine

```
$ git filter-branch --subdirectory-filter trunk HEAD  
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)  
Ref 'refs/heads/master' was rewritten
```



Filter-branch (2)

Changer une adresse email globalement

```
$ git filter-branch --commit-filter '  
    if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];  
    then  
        GIT_AUTHOR_NAME="Scott Chacon";  
        GIT_AUTHOR_EMAIL="schacon@example.com";  
        git commit-tree "$@";  
    else  
        git commit-tree "$@";  
    fi' HEAD
```



Fichier annoté

La commande **git blame** montre quel est le dernier commit qui a modifié chaque ligne d'un fichier. Cela peut être utile pour le débogage

```
$ git blame -L 12,22 simplegit.rb
```

```
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show
#{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame
#{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```




Recherche dichotomique

La commande **bisect** effectue une recherche par dichotomie dans l'historique afin d'aider à identifier aussi vite que possible quel commit a déclenché un bug

- Pour démarrer la recherche il faut lancer *git bisect start*,
- Ensuite, **git bisect bad** afin d'indiquer que le commit courant contient le bug
- Enfin avec **git bisect good [good_commit]**, on lui indique un commit où le bug n'existait pas

Git calcule le nombre de commits entre le bon et le mauvais et effectue un checkout du commit du milieu.

Il est alors possible d'exécuter les tests afin de voir si le bug existe.

- Si il existe, le bug a été introduit avant. On exécute **git bisect bad** et Git fait un checkout du commit milieu entre le commit courant et le bon commit
- Sinon, on exécute **git bisect good** et Git fait un checkout vers l'avant en utilisant la dichotomie



Dangling commit

Certains commits ne faisant plus partie d'une branche sont difficiles à retrouver.

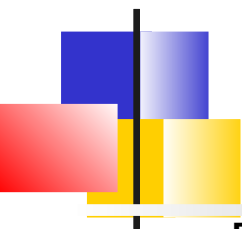
La commande ***fsck*** permet de vérifier l'intégralité des objets de dépôts et de trouver par exemple les commits n'étant plus sur une branche

```
git fsck --lostfound
```

```
dangling commit 93b0c51cfea8c731aa385109b8e99d19b38a55be
```



Personnalisation de Git



Configuration d'un client

De nombreux paramètres pouvant être positionnés par *git config* existent

- ***commit.template*** : Pointe vers un fichier contenant le gabarit des messages de commit
- ***user.signingkey*** : Permet de spécifier la clé GPG signant les tags
- ***core.excludesfile*** : Pointe vers un fichier de type *.gitignore*
- ***color.ui*** : Colorisation
- ***merge.tool*** : Outil de fusion
- ***diff.external*** = Outil de diff
- ***core.autocrlf*** : Utile lorsque des développeurs sont sous des plate-formes différentes comprenant Windows
- ***core.whitespace*** : Enlever les espaces de fin ou de début



Examples

```
$ git config --global commit.template  
  $HOME/.gitmessage.txt  
$ git config --global user.signingkey <gpg-key-id>  
$ git config --global core.exclusefiles ~/.mygitignore  
$ git config --global color.ui true  
$ git config --global diff.external extDiff  
$ git config --global core.autocrlf true  
$ git config --global core.whitespace \  
  trailing-space,space-before-tab,indent-with-non-tab
```



Hooks

Git dispose d'un moyen de lancer des scripts personnalisés quand certaines actions importantes ont lieu : les ***hooks***

Il y a deux types de *hook* :

- Côté client: ils concernent les opérations de client telles que la validation et la fusion.
- Côté serveur : Ils concernent les opérations serveur telles que la réception de commits.



Mise en place

Les crochets sont tous stockés dans le sous-répertoire ***hooks***

- Ce sous-répertoire contient par défaut des fichiers exemples (suffixés *.sample*)

Pour mettre en place un *hook*, il suffit de positionner un script avec un nom prédéfini et avec les bonnes permissions dans ce répertoire



Hooks de commit (client)

pre-commit est lancé en premier, avant la saisie du message de validation, il a pour but de vérifier ce qui va être committé. Si le script renvoie un code de sortie non null, le commit est annulé

prepare-commit-msg est appelé avant l'ouverture de l'éditeur de message, il permet d'éditer le message par défaut

commit-msg a accès au message de commit et peut annuler le commit si il renvoie non null

post-commit est exécuté après le commit, il sert à faire de la notification



Autres *hooks* client

pre-rebase est invoqué avant une opération rebase et peut interrompre le processus s'il sort avec un code d'erreur non nul

post-checkout est exécuté après une commande *checkout* réussie

post-merge s'exécute à la suite d'une commande *merge* réussie



Hooks serveur

pre-receive avant une poussée de données sur le serveur

post-receive après

update est similaire au script *pre-receive* mais s'exécute une fois par branche



Merci!!!

❖ MERCI DE VOTRE ATTENTION



Annexes



Sous-module



Sous-module

Lorsqu'un projet a une dépendance sur un autre projet, une bibliothèque externe par exemple

- que l'on désire apporter ses propres modifications
- tout en profitant des évolutions de la branche principale,

=> il faut utiliser les **sous-modules**



git submodule

Les projets externes sont ajoutés comme sous-module avec la commande ***git submodule add***

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
```

```
Initialized empty Git repository in /opt/subtest/rack/.git/
```

```
remote: Counting objects: 3181, done.
```

```
remote: Compressing objects: 100% (1534/1534), done.
```

```
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
```

```
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
```

```
Resolving deltas: 100% (1951/1951), done.
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
#       new file:   .gitmodules
```

```
#       new file:   rack
```

```
#
```



git submodule

Les projets externes sont ajoutés comme sous-module avec la commande ***git submodule add***

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .gitmodules
#       new file:   rack
#
```




.gitmodules

Le fichier ***.gitmodules*** est un fichier de configuration qui stocke la correspondance entre l'URL du projet et le sous-répertoire local

```
$ cat .gitmodules
[submodule "rack"]
    path = rack
    url = git://github.com/chneukirchen/rack.git
```

Ce fichier est versionné (comme le fichier *.gitignore*)

C'est grâce à ce fichier que les collaborateurs qui récupèrent les données de votre repository connaissent la configuration des sous-modules.



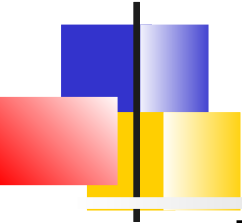
Comportement du sous-répertoire

Le sous-répertoire du sous module fait partie du répertoire de travail mais Git ne suit pas son contenu lorsque les commandes Git ne sont pas effectuées dans ce sous-répertoire.

Toutes les commandes *git* s'exécutent indépendamment dans les 2 répertoires et le répertoire du module est un commit particulier du référentiel

Lorsque l'on effectue des changements dans ce sous-répertoire, le projet parent détecte que le HEAD du module a changé et enregistre le commit courant du sous-module

Ainsi, lorsque les collaborateurs clonent le projet, il recréent exactement le même environnement

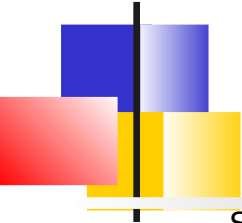


Cloner un projet avec un sous-module

Lorsque l'on clone un projet avec un sous-module, on récupère le sous-répertoire du sous-module vide.

Il faut alors exécuter deux commandes :

- ***git submodule init*** pour initialiser la configuration locale
- ***git submodule update*** pour récupérer les données et effectuer un checkout du commit référencé par le projet parent



Mise à jour

Si un autre développeur effectue des changements dans le sous-module et les commit another developer makes changes to the rack code and commits, and you pull that reference down and merge it in, you get something a bit odd:

```
$ git merge origin/master
```

```
Updating 0550271..85a3eee
```

```
Fast forward
```

```
rack | 2 +- 
```

```
1 files changed, 1 insertions(+), 1 deletions(-)
```

```
[master*]$ 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:   rack
```

```
#
```

You merged in what is basically a change to the pointer for your submodule; but it doesn't update the code in the submodule directory,

```
git submodule update --init
```



Projets arborescents

Une bonne approche pour la gestion de projets arborescents est de :

- Installer un dépôt Git indépendant pour chaque sous-répertoire du projet parent
- Puis créer le dépôt du projet parent avec des sous-modules

L'avantage de cette approche est qu'il est plus facile de définir les relations entre les projets avec des tags et des branches



Migration SVN vers Git



Introduction

Git propose un pont bidirectionnel avec *Subversion* nommé ***git svn***.

Cet outil permet d'utiliser *Git* comme client d'un serveur Subversion

=> On peut alors utiliser les fonctionnalités locales de Git puis pousser vers un serveur Subversion

Certaines opérations sont cependant à éviter lorsque l'on travaille de cette façon :

- Ne pas essayer de modifier l'historique après avoir poussé vers le serveur
- Ne pas pousser parallèlement vers un serveur *Git* que d'autres développeurs utiliseraient



Clone de dépôt

git svn clone [svnurl] permet d'importer un dépôt Subversion dans un référentiel Git local

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
    A    m4/acx_pthread.m4
    A    m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
    file:///tmp/test-svn/branches/my-calc-branch r76
```

Les tags subversion sont ajoutés comme branche distantes



Clone de dépôt

git svn clone [svnurl] permet d'importer un dépôt Subversion dans un référentiel Git local

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
    A    m4/acx_pthread.m4
    A    m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
    file:///tmp/test-svn/branches/my-calc-branch r76
```

Les tags subversion sont ajoutés comme branche distantes



Committer vers subversion

Pour pousser vers un serveur Subversion

```
$ git svn dcommit
```

```
Committing to un file:///tmp/test-svn/trunk ...
```

```
    M      README.txt
```

```
Committed r79
```

```
    M      README.txt
```

```
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
```

```
No changes between current HEAD and refs/remotes/trunk
```

```
Resetting to the latest refs/remotes/trunk
```

Cela prend tout les commits effectués et effectue un commit subversion pour chaque et réécrit le commit local pour inclure un identifiant svn : ***git-svn-id***

Cela signifie que tous les checksum SHA-1 changent



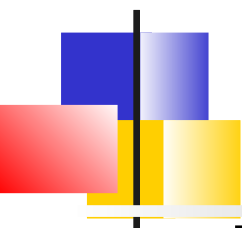
Se synchroniser avec Subversion

git svn rebase récupère les modifications du serveur que l'on ne détient pas localement et *rebase* votre travail sur le sommet de l'historique du serveur

```
$ git svn rebase
      M      README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

Il est important de se rappeler que si des modifications sont apparues sur le serveur mais qu'elles ne sont pas en conflit, l'opération *dcommit* réussira

=> Si les changements sont incompatibles mais ne sont pas en conflit, cela peut provoquer des problèmes difficiles à diagnostiquer



Utilisation des branches svn

Pour créer une nouvelle branche dans Subversion : ***git svn branch [branchname]***

Équivalent à

svn copy trunk branches/[branchname]

Par contre, cette commande ne bascule pas vers la nouvelle branche

Git détermine la branche svn où vont les *dcommits* en se basant sur le dernier *git-svn-id* de l'historique de la branche locale courante

Pour basculer sur une branche nommée *opera* :

```
$ git branch opera remotes/opera
```



Commandes *svn*

git svn fournit également des commandes similaires à subversion

\$ *git svn log* : Historique SVN (fonctionne offline et ne montre que les commits du serveur *svn*)

\$ *git svn blame [FILE]* : équivalent à *svn annotate*, Liste des changements d'un fichier

\$ *git svn info* : Informations sur le serveur

Si on clone u dépôt Subversion qui a *svn:ignore*, la commande ***git svn create-ignore*** permet de créer le fichier *.gitignore* équivalent

La commande ***git svn show-ignore*** peut également être utilisée :

```
$ git svn show-ignore > .git/info/exclude
```



Migration

git svn peut être facilement utilisé pour migrer vers un serveur Git.

Il suffit :

- de cloner le dépôt *svn* avec *git svn clone*
- D'arrêter d'utiliser le serveur *svn*
- Pousser vers un nouveau serveur *Git*

Cependant, cette technique peut rester imparfaite



Migration des utilisateurs

Pour migrer les informations d'utilisateur, il est nécessaire de mettre au point un fichier de correspondance entre les utilisateurs svn et les utilisateurs Git

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
$ git svn clone http://my-project.googlecode.com/svn/ \
    --authors-file=users.txt --no-metadata -s
    my_project
```

Cette commande supprime également l'identifiant *git-svn-id* des commits



Récupération des branches et des tags

Pour récupérer correctement les branches et les tags, on doit déplacer les étiquettes pour qu'elles deviennent de vraies étiquettes, et le reste des branches pour qu'elles deviennent locale.

Les tags :

```
$ git for-each-ref refs/remotes/tags | cut -d / -f 4- | grep  
-v @ | while read tagname; do git tag "$tagname"  
"tags/$tagname"; git branch -r -d "tags/$tagname"; done
```

Les branches

```
$ git for-each-ref refs/remotes | cut -d / -f 3- | grep -v @  
| while read branchname; do git branch "$branchname"  
"refs/remotes/$branchname"; git branch -r -d  
"$branchname"; done
```




Déclaration du référentiel distant

La dernière chose à faire est d'ajouter le nouveau serveur Git en tant que référentiel distant et à y pousser le projet transformé

```
$ git remote add origin git@my-git-server  
:myrepository.git
```

```
$ git push origin --all  
$ git push origin --tags
```