

Contrôle de versions avec Git

**Système de gestion de versions distribué
Révisions de fichiers
Modèle objets Git
Développement collaboratif**

Leuville Objects
3 rue de la Porte de Buc
F-78000 Versailles
FRANCE

tel : + 33 (0)1 39 50 2000
fax: + 33 (0)1 39 50 2015

www.leuville.com
contact@leuville.com

© Leuville Objects, 1996-2016
29 rue Georges Clémenceau
F-91310 Leuville sur Orge
FRANCE

<http://www.leuville.com>

Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Les marques citées sont des marques commerciales déposées par leurs propriétaires respectifs.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A RÉPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

Table des matières

Présentation de Git.....	1
Concepts de base du contrôle de versions	1-3
La gestion centralisée ou distribuée.....	1-5
Principe de fonctionnement	1-7
Les différentes solutions de gestion de versions.....	1-9
Installation et configuration de Git.....	13
Installation sous différents systèmes (1/2).....	2-15
Configuration de l'environnement de travail.....	2-19
Déclaration d'outils graphiques de comparaison/fusion.....	2-21
Présentation d'outils graphiques	2-23
Utilisation de Git, les fondamentaux	25
Le modèle objet Git	3-27
Le répertoire de travail et le répertoire Git	3-33
La zone d'index ou staging area	3-35
Les concepts de branche, tag et dépôt.....	3-37
Création et initialisation d'un dépôt.....	3-39
Gestion locale des fichiers.....	41
Consultation de l'état du répertoire de travail	4-43
Ajout, ignorance, modification, suppression et recherche de fichiers	4-45
Annulation et visualisation des modifications	4-49
Parcours de l'historique des révisions	4-51
Gestion des branches.....	55
La branche master	5-57
Création de branches.....	5-59
Changement de branche.....	5-61
Fusion de branches.....	5-63
Gestion des conflits.....	5-65
Partage de travail et collaboration	67
Mise en place d'un dépôt distant	6-69
Les branches distantes	6-71
Récupération des modifications.....	6-73
Publier ses modifications	6-75
Mise en oeuvre des outils Git.....	77
Git-Gui/Gitk et SourceTree : clients graphiques Git pour Windows.	7-79
TortoiseGit : l'extension Git pour l'explorateur Windows.....	7-81
GitWeb : l'interface Web de navigation au sein de dépôts Git	7-83
GitHub : service Web d'hébergement de dépôts Git	7-85
GitLab et Gogs : alternatives auto-hébergées à GitHub	7-87
Gerrit : application de revue de code	7-89

GIT

Présentation de Git

Version 1.1

- Concepts de base du contrôle de versions
- La gestion centralisée ou distribuée
- Principe de fonctionnement
- Les différentes solutions de gestion de versions
- Références

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

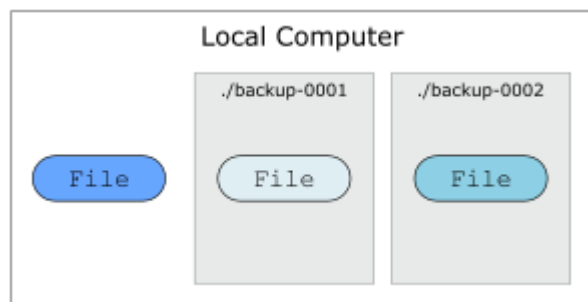
Concepts de base du contrôle de versions

Qu'est-ce que la gestion de versions ?

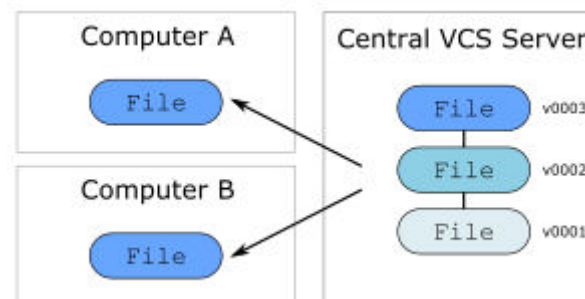
- Enregistrer l'évolution d'un ensemble de fichiers
- Identifier les modifications : date, auteur et nature
- Pouvoir ramener un fichier à un état antérieur

Systèmes de gestion de versions

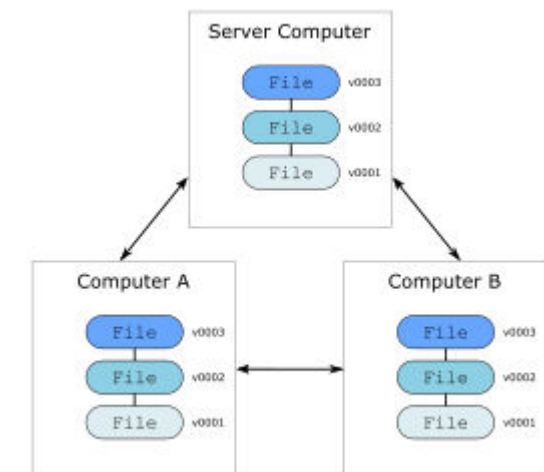
- **Local** : recopie des fichiers dans un répertoire pour sauvegarde
- **Centralisé** : un serveur central contient tous les fichiers et l'historique (non partagé avec les utilisateurs)
- **Distribué** : chaque utilisateur dispose des fichiers et de l'historique



Local Version Control System (LVCS)



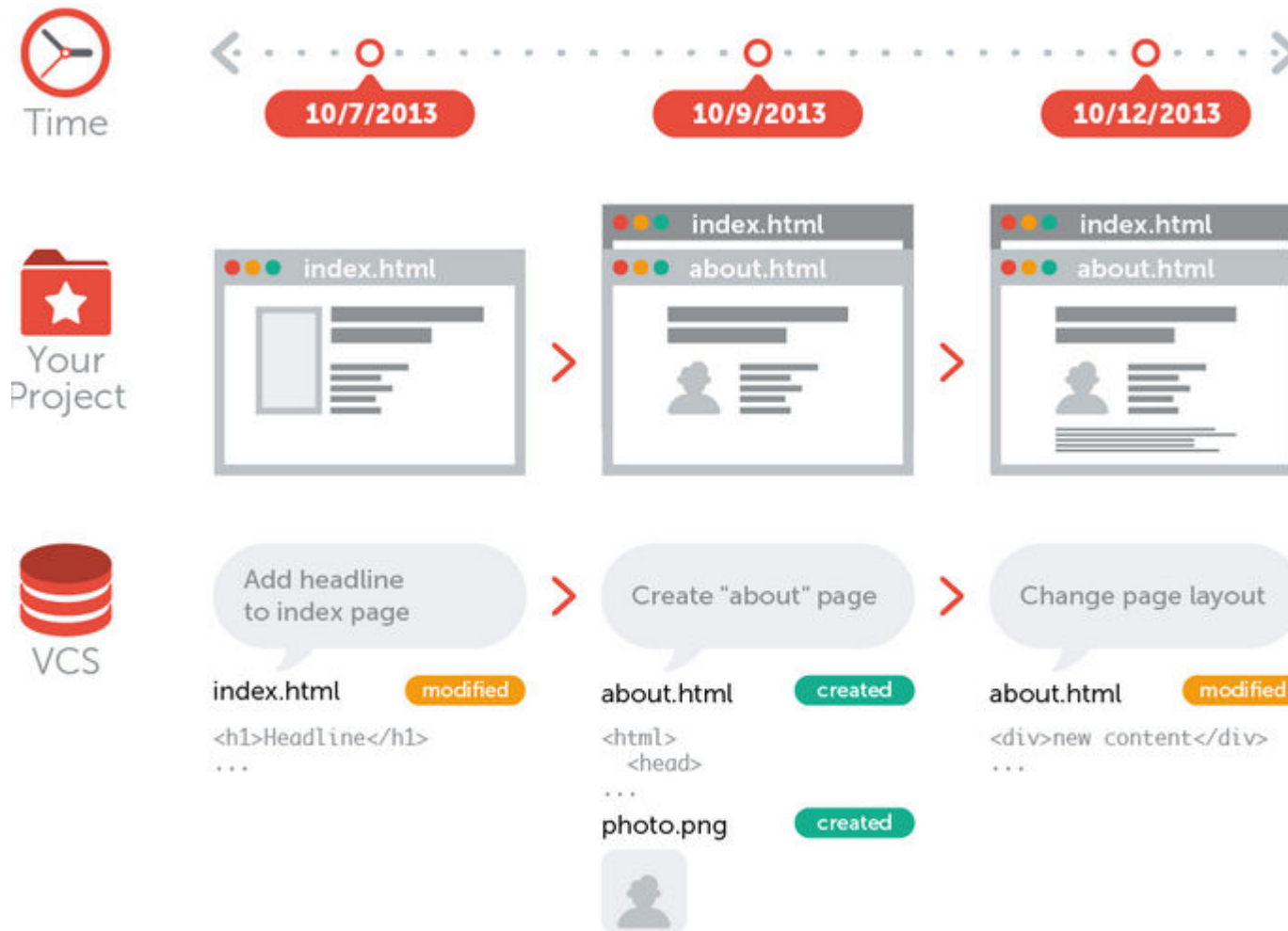
Centralized Version Control System (CVCS)



Distributed Version Control System (DVCS)

Concepts de base du contrôle de versions

Notes



Workflow basique d'un système de gestion de versions

Source : <http://www.git-tower.com/learn/ebook/command-line/basics/what-is-version-control#start>

La gestion centralisée ou distribuée

Concepts de base

- Organiser l'historique de l'état des fichiers
- Contrôler les versions locales et distantes
- Synchroniser et transférer les fichiers
- Gérer la sécurité

Gestion centralisée

- Un serveur central contient tous les fichiers et maintient l'historique
- Les utilisateurs récupèrent la dernière version des fichiers (sans l'historique)
- Workflow centralisé : la majorité des opérations nécessite de joindre le serveur
- Point de défaillance qui peut s'avérer critique (perte de l'historique)
- Annulation d'opérations difficile voir impossible

Gestion distribuée

- Les utilisateurs récupèrent tout le dépôt : ils disposent d'une copie locale du dépôt
- Possibilité de travailler hors connexion
- Opérations plus rapides car elles affectent majoritairement le dépôt local
- Suppression de la dépendance avec un serveur centralisé : le serveur sert de point de rencontre
- Rend possible certains usages comme l'annulation ou la création de branches indépendantes

La gestion centralisée ou distribuée

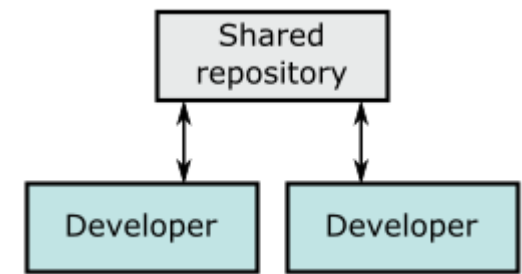
Notes

Principe de fonctionnement

- Un DVCS permet plus de flexibilité aux collaborateurs qu'un CVCS
- Les collaborateurs partagent leurs contributions selon un mode de gouvernance établi
- Le mode de gouvernance doit être adapté au nombre de collaborateurs et à leur fréquence de contribution

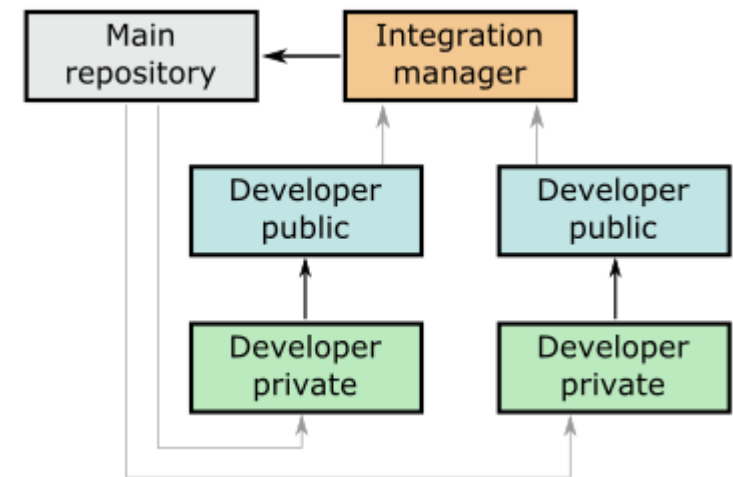
Gestion centralisée

- Mode de gouvernance semblable au fonctionnement d'un CVCS
- Un serveur est utilisé comme dépôt central
- Les développeurs "poussent" (*push*) leurs contributions sans écraser le travail des autres



Gestionnaire d'intégration

- Un serveur est utilisé comme dépôt officiel
- Seul le gestionnaire d'intégration peut écrire sur le dépôt officiel
- Un développeur a un accès en lecture au dépôt officiel qu'il clone et modifie
- Le développeur pousse ses contributions sur un dépôt public
- Il contacte le gestionnaire d'intégration pour faire intégrer son travail

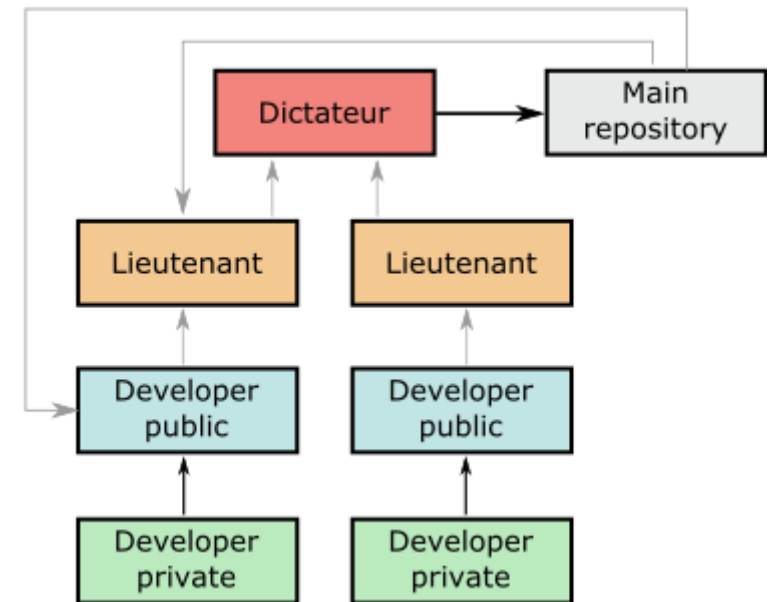


Principe de fonctionnement

Notes

Gestion par un dictateur bienveillant et ses lieutenants

- Variante du mode de gouvernance par gestionnaire d'intégration pour des gros projets
- Les lieutenants sont des gestionnaires d'intégration gérant chacun une partie du projet
- Ils intègrent le travail des développeurs dans leur propre dépôt
- Le dictateur bienveillant intègre les contributions collectées par ses lieutenants
- GNU/Linux est le projet le plus célèbre utilisant ce mode de gouvernance



Les différentes solutions de gestion de versions

Systèmes de gestion de versions locaux (LVCS)

- 1972 : Source Code Control System (SCCS)
- 1982 : Revision Control System (RCS)

Systèmes de gestion de versions centralisés (CVCS)

- 1990 : Concurrent Version System (CVS)
- 1994 : Microsoft Visual SourceSafe (VSS)
- 1995 : Perforce
- 2000 : Subversion (SVN)

Systèmes de gestion de versions distribués (DVCS)

- 2003 : ArX, Darcs, Monotone, SVK
- 2005 : Bazaar, Git, Mercurial
- 2006 : Fossil

Source : http://en.wikipedia.org/wiki/List_of_revision_control_software

Les différentes solutions de gestion de versions

Notes

Références

Pro Git

- Livre mis en avant par le site officiel de Git : sa dernière édition date de 2014
- Il peut être téléchargé gratuitement
 - Version originale en anglais :
<https://progit2.s3.amazonaws.com/en/2015-03-06-439c2/progit-en.376.pdf>
 - Version partiellement traduite en français :
<https://progit2.s3.amazonaws.com/fr/2014-12-20-17ea2/progit-fr.236.pdf>
- Le livre peut également être consulté en ligne : <http://git-scm.com/book/en/v2>

Git Community Book

- Livre écrit par la communauté Git pour un apprentissage rapide de Git
- Consultation en ligne : <https://alexgirard.com/git-book/>
- Télécharger au format PDF : <https://alexgirard.com/git-book/book.pdf>

Internet

- De nombreuses ressources sont disponibles sur Internet grâce à la popularité de Git
- Exemple de 10 cours pour débutant : <http://sixrevisions.com/resources/git-tutorials-beginners/>

Références

Notes

GIT

Installation et configuration de Git

Version 1.1

- Installation sous différents systèmes
- Configuration de l'environnement de travail
- Déclaration d'outils graphiques de comparaison/fusion
- Présentation d'outils graphiques

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Installation sous différents systèmes (1/2)

UNIX

- Utiliser le gestionnaire de paquets de votre distribution pour installer le paquet `git`
 - **Debian/Ubuntu :**
`$ apt-get install git`
 - **Fedora :**
`$ yum install git`
 - **Gentoo :**
`$ emerge --ask --verbose dev-vcs/git`
 - **Arch Linux :**
`$ pacman -S git`
 - **openSUSE :**
`$ zypper install git`
 - **FreeBSD :**
`$ cd /usr/ports/devel/git`
`$ make install`
 - **Solaris 11 Express :**
`$ pkg install developer/versioning/git`
 - **OpenBSD :**
`$ pkg_add git`

Installation sous différents systèmes

Notes

Source : <http://git-scm.com/download/linux>

Compilation de Git :

<http://git-scm.com/book/en/v2/Getting-Started-Installing-Git#idm7744576>

Installation sous différents systèmes

Windows

- Il faut télécharger le programme d'installation `msysgit` pour installer Git sur Windows
<http://msysgit.github.io>
- Le programme `msysgit` installe plusieurs composants
 - **MinGW** (Minimalist GNU for Windows) : portage pour Windows des outils de compilation GNU (GCC/GNU Compiler Collection)
 - **MSYS** (Minimal System) : composant de MinGW fournissant un terminal Unix pour Windows
 - **Git** : le système de gestion de versions distribué conçu par Linus Torvalds
- Lors de l'installation deux écrans doivent attirer notre attention
 - **Ecran 6 "Adjusting your PATH environment"** : impacte la variable d'environnement `PATH`

<i>Use Git from Git Bash only</i>	Cette option ne modifie pas le <code>PATH</code> : Git peut être utilisé uniquement depuis le terminal Unix (installé par MSYS)
<i>Use Git from the Windows Command Prompt (WCP)</i>	Cette option modifie légèrement le <code>PATH</code> : Git pourra être utilisé depuis le terminal Unix et Windows (<code>cmd.exe</code>)
<i>Use Git and optionnal Unix tools from the WCP</i>	Cette option modifie fortement le <code>PATH</code> pour pouvoir utiliser Git et d'autres outils Unix dans le terminal Windows

- **Ecran 7 "Configuring the line ending conversions"** : impacte le traitement des caractères de "retour à la ligne" (LF)

<i>Checkout Windows-style, commit Unix-style line endings</i>	Convertie les caractères LF en CRLF en entrée et CRLF en LF en sortie
<i>Checkout as-is, commit Unix-style line endings</i>	Convertie les caractères CRLF en LF en sortie uniquement
<i>Checkout as-is, commit as-is</i>	Pas de conversion des caractères de fin de ligne

Note : l'option choisie dans cette écran correspond à la valeur assignée à l'option de configuration `core.autocrlf`

Installation sous différents systèmes

Notes

A propos des caractères de "retour à la ligne"

- Sous Unix : le caractère LF (Line Feed) marque un retour à la ligne
- Sous Windows : la combinaison des caractères CRLF (Carriage Return + Line Feed) marque un retour à la ligne

Il est recommandé d'utiliser le caractère LF pour marquer un retour à la ligne plutôt que CRLF pour des projets multi-plateformes.

Configuration de l'environnement de travail

Configuration de Git

- Définir l'identité utilisée par défaut lors de vos contributions

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

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

Interface de configuration en ligne de commande

- La commande `git config` permet consulter et de définir des options de configuration
 - Les options s'appliquent à un dépôt (argument `--local` par défaut)
 - Elles s'appliquent globalement à l'utilisateur quand l'argument `--global` est renseigné
- La documentation référence toutes les options existantes
http://git-scm.com/docs/git-config.html#_variables
- Lister toutes les options de configuration

```
$ git config -l|--list
```
- Lire une option de configuration

```
$ git config [--global|--local|--system] option.name
```
- Définir une option de configuration

```
$ git config [--global|--local|--system] option.name value
```
- La commande `git help` permet d'afficher le manuel d'utilisation

Configuration de l'environnement de travail

Notes

Activer la couleur dans Git pour améliorer la lisibilité des messages Git dans la console

```
$ git config --system color.ui
```

```
$ git config --system color.branch auto
```

```
$ git config --system color.diff auto
```

```
$ git config --system color.status auto
```

L'utilisation de l'argument `--system` applique les options à tous les utilisateurs

Fichier de configuration `.gitconfig`

Les options de configuration de Git sont stockées dans un fichier `.gitconfig`

Ce fichier peut exister à différents niveaux :

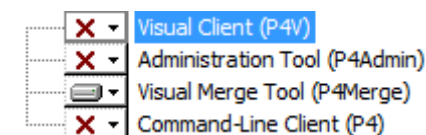
- Système dans `/etc/gitconfig`
 - Utilisateur dans `~/.gitconfig` (ou `~/.config/git/config`)
 - Local (au niveau d'un dépôt) dans `.git/config`
-

Déclaration d'outils graphiques de comparaison/fusion

- La comparaison et la fusion de fichiers s'effectuent dans la console par défaut
- Il est possible d'installer des outils graphiques pour faciliter ces opérations

Meld et P4Merge

- Installer des outils de comparaison et de fusion
 - **Meld** : <http://meldmerge.org/>
 - **P4Merge** : http://www.perforce.com/downloads/20-User?qt-perforce_downloads_step_3=1#product-detail-10
Sous Windows, veuillez à installer P4Merge uniquement



P4Merge pour Windows

Configuration des outils sous Git

- Renseigner les nouveaux outils


```
$ git config --global mergetool.meld.path 'C:\\Program Files (x86)\\Meld\\Meld.exe'
```

```
$ git config --global mergetool.p4merge.path 'C:\\Program Files\\Perforce\\p4merge.exe'
```
- Définir l'outil utilisé par défaut


```
$ git config --global merge.tool meld
```
- La comparaison avec `git difftool` et la fusion avec `git mergetool` lancent le programme défini par défaut
- L'argument `-t|--tool` permet de spécifier le programme à lancer


```
$ git difftool -t p4merge 2f31b5f cf3c35f
```

Déclaration d'outils graphiques de comparaison/fusion

Notes

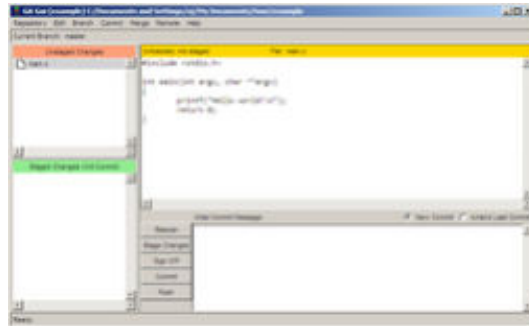
Le fichier de configuration de l'utilisateur `~/.gitconfig` peut directement être édité pour modifier les options de configuration

```
[user]
  name = Steven Enten
  email = steven@enten.fr
[merge]
  tool = meld
[mergetool "p4merge"]
  path = C:\\Program Files\\Perforce\\p4merge.exe
[mergetool "meld"]
  path = C:\\Program Files (x86)\\Meld\\Meld.exe
```

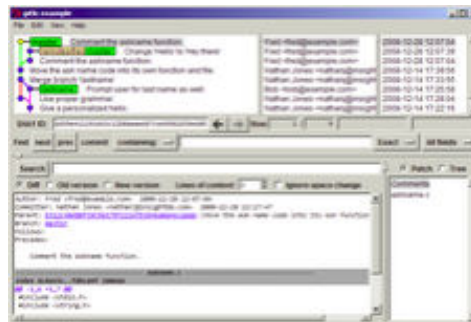
Les fichiers de configuration des autres niveaux (système et local) peuvent également être édités

Présentation d'outils graphiques

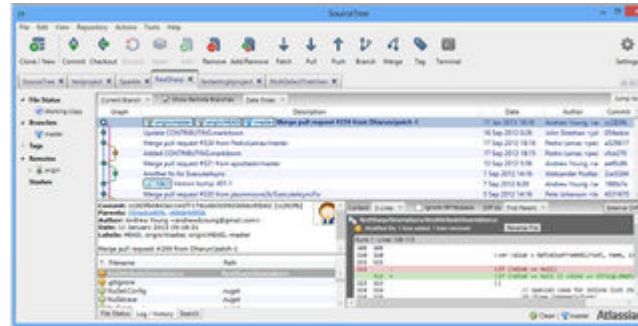
git-gui Linux/Windows



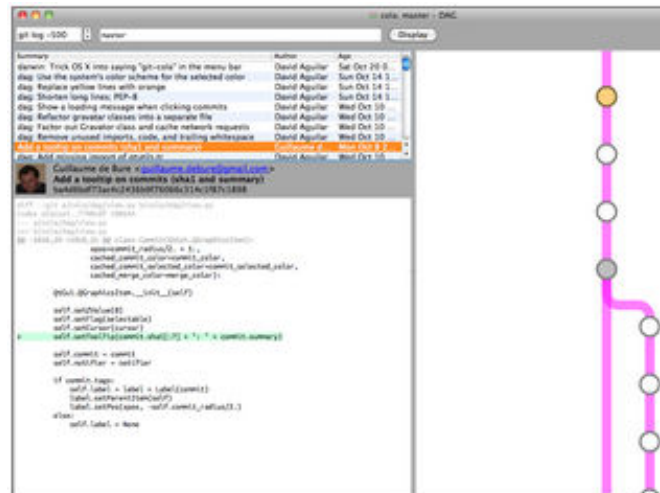
gitk Linux/Windows



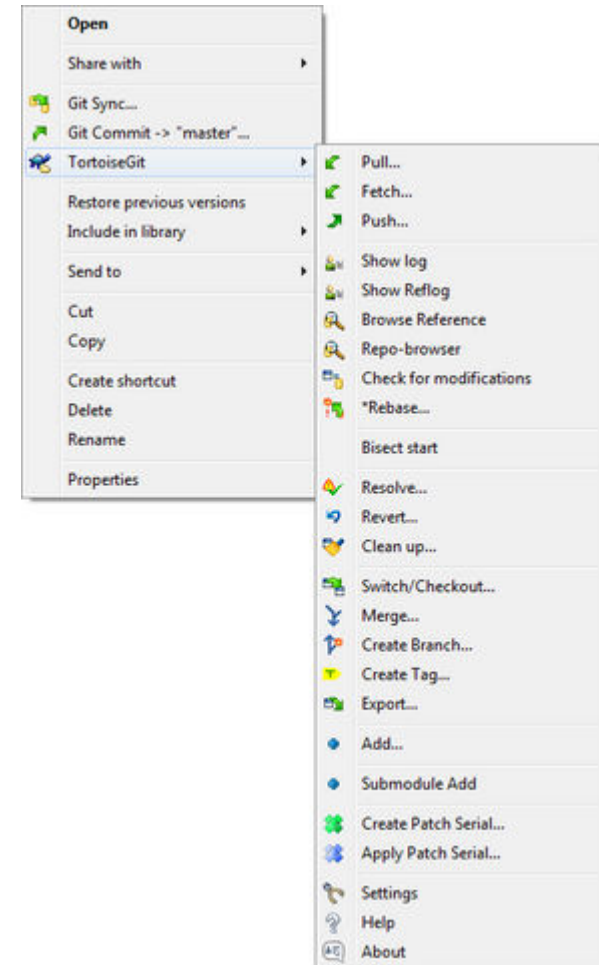
SourceTree Windows



git-cola Linux/Mac/Windows



TortoiseGIT Windows



Présentation d'outils graphiques

Notes

Utilisation de Git, les fondamentaux

- Le modèle objet Git : blob, tree, commit et tag
- Le répertoire de travail et le répertoire Git
- La zone d'index ou staging area
- Les concepts de branche, tag et dépôt
- Création et initialisation d'un dépôt

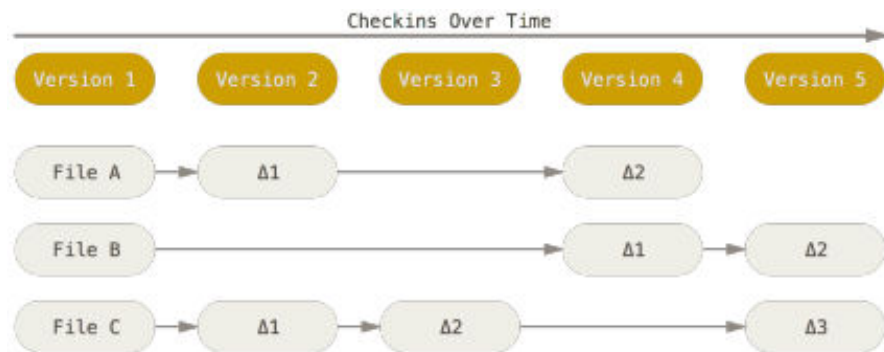
(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

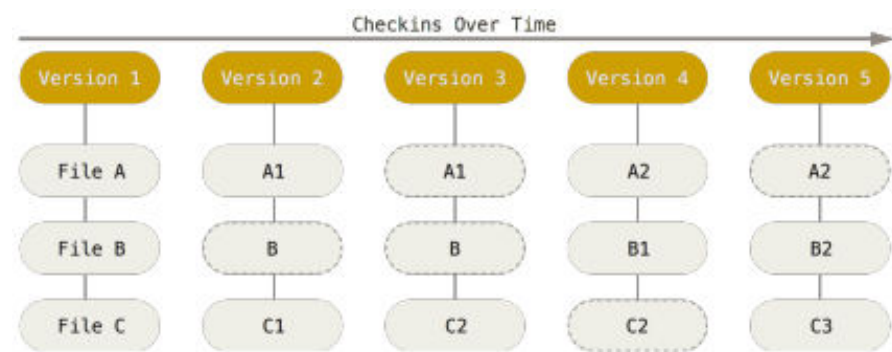
Le modèle objet Git

Base conceptuelle

- La majorité des VCS construisent l'historique comme une liste de modifications apportées aux fichiers (cas de Bazaar, CVS, Perforce et SVN par exemple)
- Git construit l'historique en créant des "images" (*snapshots*) de l'état des fichiers et en stockant des références vers celles-ci (ainsi un fichier n'est jamais dupliqué)



VCS gérant l'historique comme une liste de modification de fichiers



Git gérant l'historique comme un flux de snapshots

- Note : l'historique d'un VCS est considérée comme une base de données
- Anecdote : lorsque Mozilla a migré ses dépôts de SVN vers Git, la taille occupée est passée de 12 GB à 420 MB*
Source : https://git.wiki.kernel.org/index.php/GitSvnComparison#Smaller_Space_Requirements

Le modèle objet Git

Notes

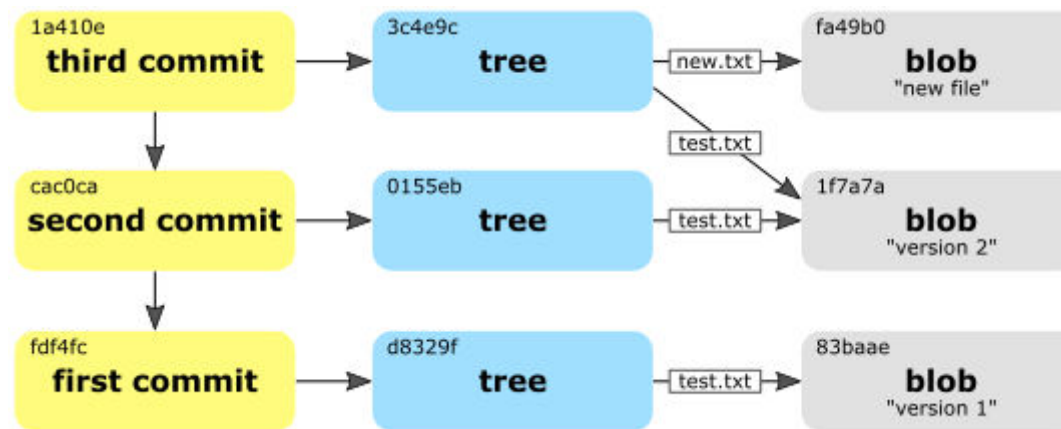
Le modèle objet Git

Empreinte SHA-1

- Chaque information de l'historique est vérifiée et référencée par une somme de contrôle de 40 caractères hexadécimaux
- Une somme de contrôle représente l'empreinte (*hash*) SHA-1 d'un fichier ou d'un dossier référencé
- Exemple d'empreinte SHA-1 : 24b9da6552252987aa493b52f8696cd6d3b00373
- Cette technique a de nombreux avantages
 - Git peut rapidement savoir si deux objets sont les mêmes (en comparant les empreintes)
 - Le même contenu stocké dans des dépôts différents sera toujours stocké avec la même empreinte
 - Détection d'erreur simplifiée en comparant le nom d'une référence et l'empreinte SHA-1 du contenu référencé
- Les sommes de contrôle rendent impossible la modification de contenu sans que Git s'en rende compte

Le modèle objet Git

Notes

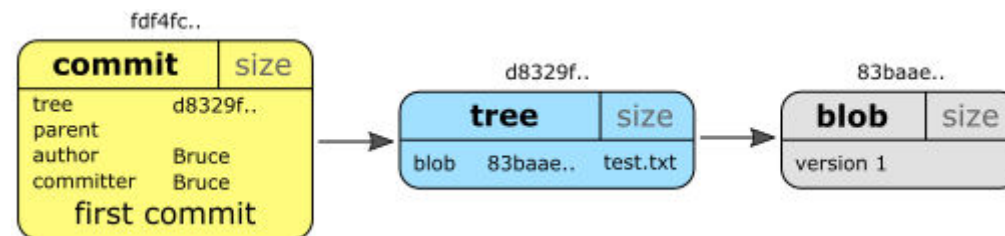


Toutes les informations sont référencées par une empreinte SHA-1

Le modèle objet Git

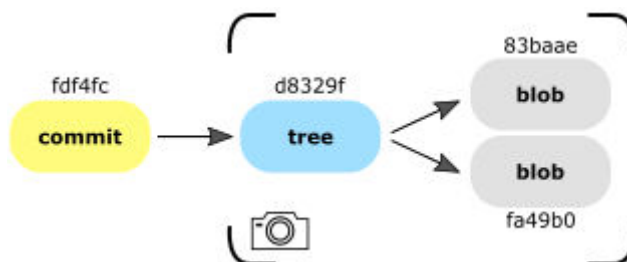
Les Objets

- Un **objet** est composé de 3 éléments
 - son **type**
 - sa **taille** : égale à la taille de son contenu
 - son **contenu** (dépend du type)



L'historique Git est une simple base clé/valeur

- Il existe 4 types d'objets différents
 - blob** : fichier binaire stockant le contenu d'un fichier
 - tree** : liste de références vers d'autres objets de type **blob** et/ou **tree** (représente un répertoire)
 - commit** : référence vers un unique **tree** représentant le projet à un certain point dans le temps
 - tag** : représente un **commit** particulier en tant que version spécifique du projet



Un commit est une "photographie" de l'état d'un tree

Le modèle objet Git

Notes

L'ensemble des objets peut se voir comme un mini-système de fichier qui se situe au-dessus du système de fichier fourni par le système d'exploitation.

Les objets sont stockées dans le répertoire `objects` du répertoire Git

- Un objet est identifié par une empreinte SHA-1 de 40 caractères
 - Les objets stockées sont répartis dans des répertoires nommé avec les 2 premiers caractères de l'empreinte et le nom du fichier de l'objet est nommé avec les 38 autres caractères
 - Exemple : un objet portant l'empreinte `d670460b4b4aece5915caf5c68d12f560a9fe3e4` est stocké dans `.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4`
-

Le répertoire de travail et le répertoire Git

Le répertoire de travail

- Contient la version courante des fichiers d'un dépôt
- Les fichiers d'un répertoire de travail évoluent
 - Ils sont directement modifiés jusqu'au prochain *commit*
 - Certaines opérations de Git modifient les fichiers du répertoire (changement de branche par exemple)
- L'historique d'un dépôt est stocké dans le répertoire `.git`

Le répertoire Git

- Répertoire contenant tous les objets Git et méta-informations : c'est l'**historique d'un dépôt**
- Ce répertoire se nomme `.git` (par défaut)
- La présence de ce répertoire indique un dépôt géré par Git
- Ce dossier est souvent considéré comme une base de données clé/valeur

```
steven@midgar:~/repo$ tree --dirsfirst -L 1 .git/
.git/
├── branches
├── hooks
├── info
├── logs
├── objects
├── refs
├── COMMIT_EDITMSG
├── config
├── description
├── HEAD
└── index

6 directories, 5 files
```

Exemple du contenu d'un repertoire .git

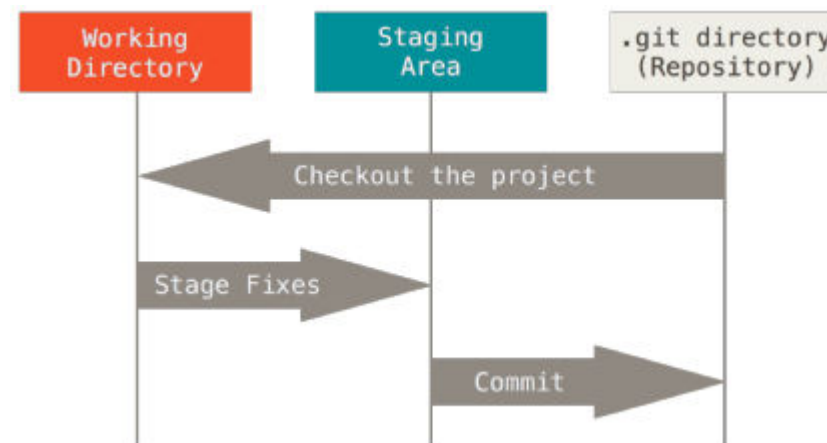
Le répertoire de travail et le répertoire Git

Notes

La zone d'index ou staging area

Les états d'un fichier

- Chaque fichier du répertoire de travail suivi avec Git (versionné) possède un état
- Git reconnaît 3 états des fichiers suivis
 - **Modifié** (*modified*) : fichier modifié mais pas encore validé en base (intégré à l'historique)
 - **Indexé** (*staged*) : fichier modifié et marqué pour faire partie de la prochaine validation (*commit*)
 - **Validé** (*committed*) : indique que le fichier est validé en base (stockée en sécurité dans l'historique)



Workflow de l'utilisation de Git

L'index

- Fichier (nommé `index`) représentant une zone d'assemblage entre le répertoire de travail et le dépôt Git
- Lors de l'utilisation de Git, l'utilisateur...
 - Modifie des fichiers dans le répertoire de travail
 - Indexe les fichiers modifiés pour les ajouter à la zone d'index pour être validé en base
 - Valide (*commit*) les fichiers indexés pour les intégrer en base (dans le répertoire `.git`)
- Ces opérations ont un impact sur le contenu de l'index

La zone d'index ou staging area

Notes

La commande `git status` permet de consulter l'état de l'index

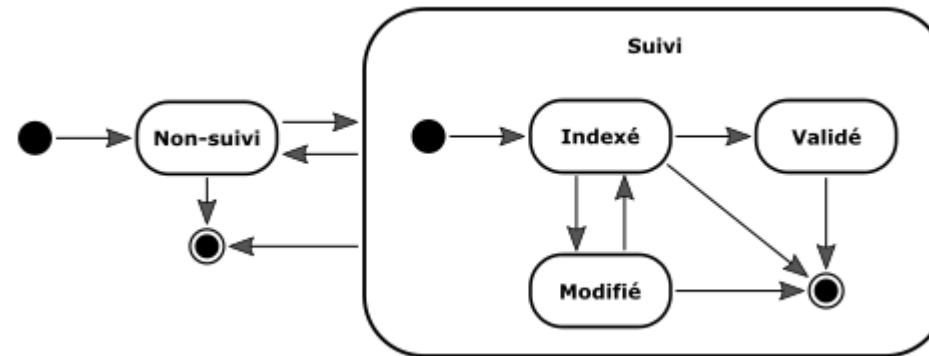


Diagramme d'états d'un fichier créé dans le répertoire de travail

Les concepts de branche, tag et dépôt

Les références

- Une référence est un fichier contenant une empreinte SHA-1 pointant vers un *commit*
- Elle permet d'identifier avec un nom clair un pointeur vers un *commit* particulier
- Git distingue **2 types de références**
 - Branche : pointeur mobile correspondant à un avancement divergent du projet (`master` par défaut)
 - Tag : pointeur figé correspondant à une version du projet
- Les références sont stockées dans le répertoire `.git/refs`

Les branches

- Pointeur automatiquement mis à jour (mobile) afin de correspondre au dernier *commit* réalisé
- La création d'une branche nécessite de renseigner un *commit* vers lequel pointer
- Le développement par branches est recommandé : c'est une fonctionnalité célèbre de Git

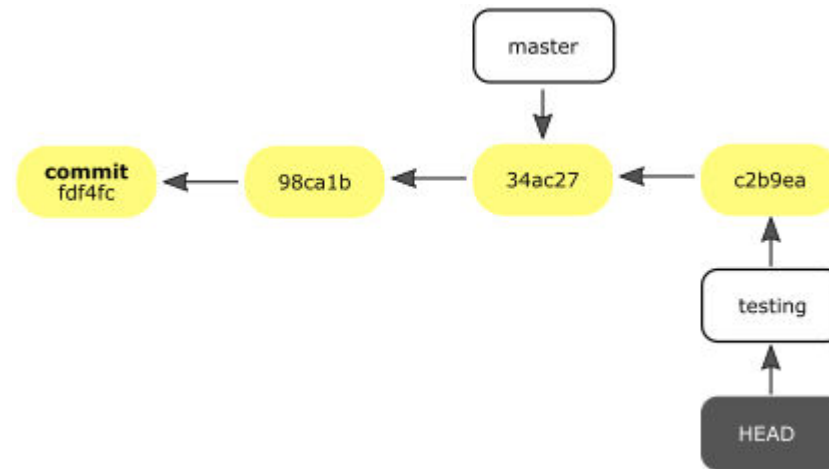
Les tags

- Pointeur fixe utilisé pour marquer un état important du projet : généralement une version (*release*)
- Git distingue les tags légers (alias de *commits*) et annotés (réels objets dans la base)

Les concepts de branche, tag et dépôt

Notes

La super référence HEAD est un **pointeur vers la branche courante**



Dépôt avec deux branches

La commande `git branch` permet de créer des branches

La commande `git tag` permet de créer des tags

Création et initialisation d'un dépôt

- L'initialisation d'un dépôt crée une base Git (`.git`)
- La commande `git init` permet d'initialiser un dépôt Git
- La commande `git clone` permet de copier un dépôt existant

Workflow local

- Indexer des fichiers pour être validé en base avec la commande `git add`
- Consulter l'état de l'index avec la commande `git status`
- Valider les fichiers en base avec la commande `git commit`
- Créer des branches avec la commande `git branch` et changer de branche avec `git checkout`
- Fusionner une branche avec la commande `git merge`

```
$ cd ~/project
$ git init                                # initialisation du dépôt
$ git add *.c README                      # ajout des fichiers C et README à l'index
$ git commit -m "initial commit"          # validation des données dans la base Git
$ git checkout -b testing                 # changement sur une nouvelle branche
# Modification des fichiers de code source C
$ git add *.c
$ git commit -m "add awesome feature"
$ git checkout master                     # retour sur la branche principale
$ git merge testing                       # fusion de la branche testing
```

Exemple de workflow local avec Git

Création et initialisation d'un dépôt

Notes

Format du message de validation (*commit*)

Des messages clairs facilitent la collaboration entre développeurs. Il faut s'habituer à soigner chaque message de validation selon quelques règles :

- Format simplifié = 1 ligne avec une brève description (50 caractères maximum)
 - Format détaillé = Format simplifié + 1 ligne vide + 1 ligne avec le message détaillé
 - Le message détaillé doit expliquer la motivation de la modification et le nouveau comportement
 - Décrire les actions sous forme d'ordres
-

Court résumé des modifications (50 caractères ou moins)

Explication plus détaillée, si nécessaire. Retour à la ligne vers 72 caractères. Dans certains contextes, la première ligne est traitée comme le sujet d'un e-mail et le reste comme le corps. La ligne vide qui sépare le titre du corps est importante (à moins d'omettre totalement le corps). Des outils tels que rebase peuvent être gênés si vous les laissez collés.

Paragraphe supplémentaire après des lignes vides.

- Les listes à puce sont aussi acceptées
- Typiquement, un tiret ou un astérisque précédés d'un espace unique séparés par des lignes vides mais les conventions peuvent varier

Edemple de message de commit valide

Git

Gestion locale des fichiers

Version 1.1

- Consultation de l'état du répertoire de travail
- Ajout, ignorance, modification, suppression et recherche de fichiers
- Annulation et visualisation des modifications
- Parcours de l'historique des révisions

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Consultation de l'état du répertoire de travail

- L'état du répertoire de travail dépend de la base Git et de la zone d'index
- La commande `git status` affiche l'état du répertoire de travail

```
steven@midgar:~/repo$ git status
Sur la branche master

Validation initiale

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)

    nouveau fichier: VERSION

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

    README
```

Etat du répertoire détaillé

```
steven@midgar:~/repo$ git status -s
A  VERSION
?? README
```

Etat simplifié du répertoire

- L'état ci-dessus véhicule différentes informations
 - Le nom de branche actuelle (`master`) sur lequel est basé le répertoire de travail
 - Les fichiers du dépôt validés lors de la prochaine validation (*commit*)
 - Les fichiers du dépôt non-suivis

Consultation de l'état du répertoire de travail

Notes

Ajout, ignorance, modification, suppression et recherche de fichiers

Ajout de fichiers

- La présence d'un fichier dans le répertoire de travail ne signifie pas qu'il est suivi par Git
- Un fichier créé n'est donc pas automatiquement ajouté à la base Git
- Il faut l'ajouter à l'index avec `git add` puis le valider avec `git commit` pour l'intégrer à la base Git
- La commande `git reset HEAD filename` permet de désindexer un fichier de la branche courante

Modification de fichiers

- La modification d'un fichier suivi par Git ne garantit pas qu'il sera mis à jour dans la base Git au prochain *commit*
- Un fichier modifié suivi par Git passe à l'état *Modified*
- Il faut de nouveau appeler `git add` pour que les modifications puissent être intégrées à la base

Suppression de fichiers

- La suppression d'un fichier du répertoire de travail n'annule pas son suivi par Git
- Il s'agit d'une modification à ajouter à l'index puis à valider (*commit*)
- La commande `git rm` permet de supprimer un fichier du répertoire de travail et à mettre à jour l'index
Note : l'option `--cached` indique de supprimer le fichier uniquement de l'index (ne plus suivre le fichier)

Ajout, ignorance, modification, suppression et recherche de fichiers

Notes

Les commandes `git reset` et `git checkout` permettent de récupérer un fichier supprimé non-validé

La commande `git diff` permet de comparer les fichiers du répertoire de travail avec d'autres révisions

La commande `git help [command]` permet d'obtenir de l'aide sur une commande

Ajout, ignorance, modification, suppression et recherche de fichiers

Ignorance de fichiers

- Il est fréquent de ne pas vouloir suivre certains fichiers d'un projet (fichiers temporaires, compilés, etc)
- Pour cela, un néophyte se garde d'ajouter ce type de fichier à l'index ou d'utiliser `git add -a|--all`
- Git permet d'ignorer certains fichiers via la création d'un fichier `.gitignore`
 - Les règles d'un `.gitignore` s'appliquent au répertoire où il est placé ainsi qu'au sous-répertoire
 - Chaque ligne du fichier décrit un motif pour ignorer des fichiers
 - Documentation en ligne : <http://git-scm.com/docs/gitignore>

Recherche de fichiers

- Git fournit des commandes pour assister la recherche de fichiers
- La commande `git ls-files` affiche des informations sur les fichiers de l'index et du répertoire de travail
 - L'option `-c` par défaut affiche les fichiers suivis, l'option `-m` affiche les fichiers à l'état *Modified*, etc
- La commande `git ls-tree` liste le contenu d'un objet tree
- La commande `git grep` permet de chercher dans les fichiers indexer dans différentes versions du dépôt
 - Exemple pour rechercher dans la révision précédente : `git grep foo HEAD^`

Ajout, ignorance, modification, suppression et recherche de fichiers

Notes

Le fichier `.git/info/exclude` est comparable à un `.gitignore` globale à tout le dépôt

Exemple de fichier `.gitignore`

```
# ignore objects and archives, anywhere in the tree.  
*.[oa]  
# ignore generated html files,  
*.html  
# except foo.html which is maintained by hand  
!foo.html
```

Annulation et visualisation des modifications

Opérations d'annulation

- Retirer un fichier ajouté à l'index
`$ git reset HEAD -- file1`
- Modifier le message du dernier *commit*
`$ git commit --amend`
- Annuler le dernier *commit* SANS restauration des fichiers modifiés dans l'état du *commit*
`$ git reset HEAD^`
- Annuler le dernier *commit* AVEC restauration des fichiers modifiés dans l'état du *commit*
`$ git reset --hard HEAD^`
- Annuler les modifications d'un fichier
`$ git checkout [tree-ish] -- file1`
- Créer un *commit* pour annuler le dernier *commit*
`$ git revert HEAD^`

Visualisation des modifications

- La consultation des logs est le meilleur moyen de visualiser les modifications apportées
 - La commande `git log` affiche le log des *commits*
 - La commande `git reflog` affiche le log des modifications de la branche courante

Annulation et visualisation des modifications

Notes

La commande `git reflog` permet de revenir facilement sur l'annulation d'un *commit* (annuler une annulation de *commit*).

```
steven@midgar:~/repo$ git log --oneline
9f7fba4 update to v0.0.2
eab16cc add readme file
820b2df initial commit
steven@midgar:~/repo$ git reflog
9f7fba4 HEAD@{0}: commit: update to v0.0.2
eab16cc HEAD@{1}: commit: add readme file
820b2df HEAD@{2}: checkout: moving from master to testing
820b2df HEAD@{3}: commit (initial): initial commit
steven@midgar:~/repo$ git reset HEAD^^
Modifications non indexées après reset :
M    VERSION
steven@midgar:~/repo$ git log --oneline
820b2df initial commit
steven@midgar:~/repo$ git reflog
820b2df HEAD@{0}: reset: moving to HEAD^^
9f7fba4 HEAD@{1}: commit: update to v0.0.2
eab16cc HEAD@{2}: commit: add readme file
820b2df HEAD@{3}: checkout: moving from master to testing
820b2df HEAD@{4}: commit (initial): initial commit
```

Dans l'exemple ci-dessus, les deux derniers *commits* ont été annulés : c'est la dernière ligne du `reflog` qui nous donne cette information.

Pour revenir au commit "*update to v0.0.2*", nous disposons de sa référence.

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

Avec cette commande, la tête du dépôt est de nouveau sur le dernier *commit*

```
steven@midgar:~/repo$ git log --oneline
9f7fba4 update to v0.0.2
eab16cc add readme file
820b2df initial commit
```

Parcours de l'historique des révisions

Usage

- Consulter l'historique ne veut pas dire chercher manuellement dans le répertoire `.git`
- Il s'agit d'afficher la liste des révisions (*commits*)
- On utilise la commande `git log` et ses nombreuses options pour afficher l'historique

```
$ git log v2.5..           # commits depuis (non-visible depuis) v2.5
$ git log test..master    # commits visibles depuis master mais pas test
$ git log master..test    # commits visibles depuis test mais pas master
$ git log master...test   # commits visibles pour test ou master,
                          #   mais pas les 2
$ git log --since="2 weeks ago" # commits des 2 dernières semaines
$ git log Makefile        # commits qui modifient le Makefile
$ git log fs/             # commits qui modifient les fichiers sous fs/
$ git log -S'foo()'       # commits qui ajoutent ou effacent des
                          #   données contenant la chaîne 'foo()'
$ git log --no-merges      # ne pas montrer les commits de merge
```

Exemples d'utilisation de la commande `git log`

- Documentation de la commande `git log` : <http://git-scm.com/docs/git-log>

Parcours de l'historique des révisions

Notes

Parcours de l'historique des révisions

Statistiques

- L'option `--stat` affiche des informations sur les fichiers modifiés par chaque *commit*

```
steven@midgar:~/repo$ git log --stat README
commit eab16cc06e2794c9489c0552e2e82cc052d12d2e
Author: Steven Enten <steven@enten.fr>
Date:   Wed Mar 11 18:03:17 2015 +0100

    add readme file

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

Formatage

- L'option `--pretty=<value>` formate l'affichage du log
- Git reconnaît différentes valeurs : `online`, `short`, `medium`, `full`, `fuller`, `email` et `raw`
- Un format personnalisé peut être défini via les nombreuses options de formatage
Cf. Section *Pretty Formats* de la documentation de `git log` : http://git-scm.com/docs/git-log#_pretty_formats

```
steven@midgar:~/repo$ git log --pretty=format:'%h par %an, %ar: %s'
9f7fba4 par Steven Enten, il y a 21 heures: update to v0.0.2
eab16cc par Steven Enten, il y a 21 heures: add readme file
820b2df par Steven Enten, il y a 21 heures: initial commit
```

Graphe

- L'option `--graph` affiche le log sous forme de graphe (représente les lignes de l'historique)

```
steven@midgar:~/repo$ git log --graph --pretty=oneline
* 060de619d07e4d2b312b1ea40f5375eefc58b73e update to v0.0.3
* 9029a7a95f41ecd5cb8ed124d82a9352194cbcd1 Merge branch 'testing'
|
* 9f7fba4b554d481234fd47d4132ff5bb3129e267 update to v0.0.2
* eab16cc06e2794c9489c0552e2e82cc052d12d2e add readme file
* | f62a7c6993e0fb43a473383db6a7d09648c3ff11 add index.html file
|/
* 820b2df9d8431eaacbcdabbae0e08e4a53fb04dd2 initial commit
```

Ordre

- Git affiche par défaut les commits dans l'ordre antichronologique : c'est l'option `--date-order`
- L'option `--topo-order` ordonne le log de manière à afficher les commits descendants avec leurs parents
- L'option `--reverse` inverse l'ordre du log (**Attention !** Option non-compatible avec `--graph`)

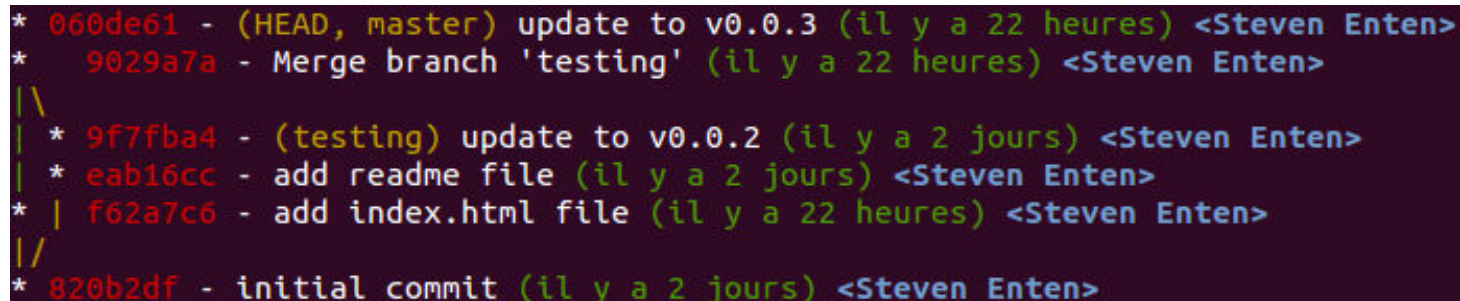
Parcours de l'historique des révisions

Notes

Log lisible avec arborescence en couleur

Commande Git

```
git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s
%C(green) (%cr) %C(bold blue)<%an>%Creset' --abbrev-commit
```



```
* 060de61 - (HEAD, master) update to v0.0.3 (il y a 22 heures) <Steven Enten>
* 9029a7a - Merge branch 'testing' (il y a 22 heures) <Steven Enten>
|\
| * 9f7fba4 - (testing) update to v0.0.2 (il y a 2 jours) <Steven Enten>
| * eab16cc - add readme file (il y a 2 jours) <Steven Enten>
* | f62a7c6 - add index.html file (il y a 22 heures) <Steven Enten>
|/
* 820b2df - initial commit (il y a 2 jours) <Steven Enten>
```

Alias `git lg` pour appeler cette commande

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

Source : <http://tilap.net/git-logs-lisibles-arborescence-couleur>

Git

Gestion des branches

Version 1.1

- La branche *master*
- Création de branches
- Changement de branche
- Fusion de branches
- Gestion des conflits

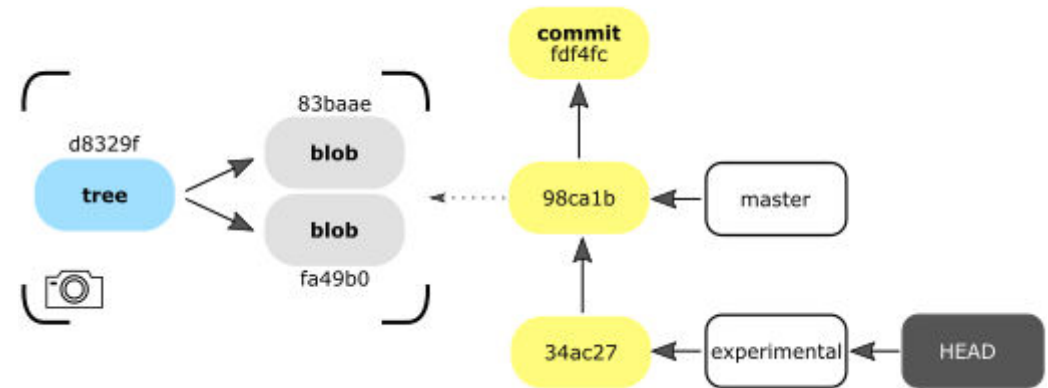
(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

La branche *master*

Rappels

- Un *commit* contient des références/pointeurs vers le ou les *commits* qui le précèdent
- Pas de parent pour la racine, un parent pour un *commit* normal et plusieurs pour un *commit* de fusion
- L'historique des *commits* forme une arborescence où chaque noeud représente un état dans le temps



Dépôt avec deux branches divergeantes

Les branches

- Une branche est un pointeur mobile vers un *commit* : il est automatiquement déplacé pour référencer le dernier *commit*
- Le super pointeur HEAD pointe sur la branche courante
- La branche *master* est la branche créée par défaut lors de l'initialisation du dépôt avec `git init`
- Par convention la branche *master* désigne la branche principale
- Les releases stables sont généralement issues de la branche *master*

La branche *master*

Notes

Création de branches

Les branches existantes

- La commande `git branch` sans option (ou avec l'option `--list`) liste les branches locales existantes
- L'option `--remotes` (ou `-r`) liste les branches distantes
- L'option `--all` (ou `-a`) liste les branches locales et distantes
- Il est possible d'afficher le *commit* pointé par chaque branche avec l'option `--verbose` (ou `-v`)

```
steven@midgar:~/repo$ git branch -v
* experimental 9f7fba4 update to v0.0.2
master         f62a7c6 add index.html file
```

Création

- La commande `git branch <name> [start-point]` crée une nouvelle branche
- Par défaut, la création d'une branche crée un nouveau pointeur sur le *commit* courant (HEAD)
- Il est possible de créer une branche pointant sur un *commit* spécifique (argument `start-point`)
- Création d'une branche divergeante de la branche *master* : `git branch experimental master`

Modification

- L'option `--move` (ou `-m`) permet de renommer une branche
- L'option `--delete` (ou `-d`) permet de supprimer une branche (`-D` pour supprimer une branche non-fusionnée)

Création de branches

Notes

Changement de branche

Modification de la branche courante

- La commande `git checkout <branchname>` permet de changer de branche (modifie le super pointeur HEAD)
- L'option `-b` créer la branche si elle n'existe pas : un *commit* spécifique (`start-point`) peut être passé en paramètre
- Exemple : `git checkout -b v.0.1.0-beta experimental`

Restauration de fichiers

- La commande `git checkout <branchname> -- <path>...` permet de restaurer l'état d'un fichier
- `<branchname>` peut être le nom d'une branche, HEAD, une référence vers un *commit* ou une référence relative
- Restaurer tous les fichiers du *commit* parent : `git checkout HEAD^ -- .`

Référence relative

- Une référence relative est construite à l'aide de caractères accolés à une référence
 - Le caractère `^` référence un ancêtre : `HEAD^2` (grand parent)
 - Le caractère `~` indique le nombre de *commits* antérieurs : `master~3`
 - Le caractère `@` permet d'indiquer une date entre accolades : `9e6fc3@{yesterday}`

Changement de branche

Notes

Commit-ish/Tree-ish	Examples
1. <sha1>	dae86e1950b1277e545cee180551750029cfe735
2. <describeOutput>	v1.7.4.2-679-g3bee7fb
3. <refname>	master, heads/master, refs/heads/master
4. <refname>@{<date>}	master@{yesterday}, HEAD@{5 minutes ago}
5. <refname>@{<n>}	master@{1}
6. @{<n>}	@{1}
7. @{-<n>}	@{-1}
8. <refname>@{upstream}	master@{upstream}, @{u}
9. <rev>^	HEAD^, v1.5.1^0
10. <rev>~<n>	master~3
11. <rev>^{<type>}	v0.99.8^{commit}
12. <rev>^{ }	v0.99.8^{ }
13. <rev>^{/<text>}	HEAD^{/fix nasty bug}
14. :/<text>	:/fix nasty bug
Tree-ish only	Examples
15. <rev>:<path>	HEAD:README.txt, master:sub-directory/
Tree-ish?	Examples
16. :<n>:<path>	:0:README, :README

Fusion de branches

Fusion

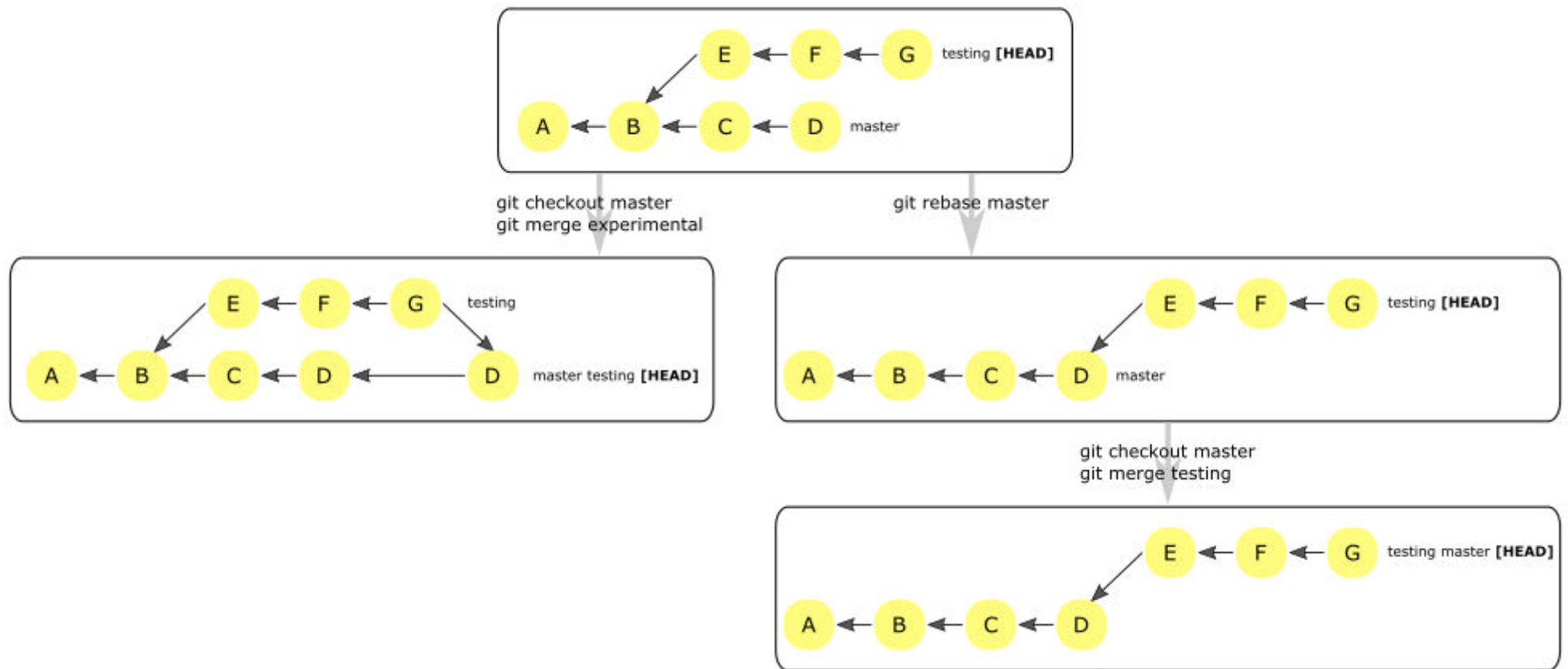
- Git encourage le développement à travers des branches thématiques (*bug*, *feature*, optimisation, etc)
- Une branche en phase finale (code vérifié) est prête à être fusionnée à la branche *master*
- La commande `git merge <branchname>` fusionne la branche courante avec la branche passée en paramètre
 - *Fast-forward* : si la branche courante n'a pas avancée par rapport à la branche fusionnée
 - *Commit* : si la branche courante a avancée par rapport à la branche fusionnée
- Des modifications peuvent être nécessaires avant la fusion en cas de conflits

Rebasage

- Le rebasage permet d'ajuster l'historique : le mettre à jour, le ré-écrire et le linéariser
- La commande `git rebase` permet de retravailler l'historique avant de partager les modifications sur un dépôt distant
- Combiner `git rebase` avec `git merge` permet de linéariser l'historique
- Le rebasage peut être source de conflits (tout comme la fusion)

Fusion de branches

Notes



Différence entre la fusion (merge) et le rebasage (rebase)

Gestion des conflits

Comparaison de branches

- La commande `git diff [ref] <ref> [--] [path]...` permet de comparer des branches et des *commits*
- Comparaison avec le dernier commit : `git diff [HEAD]`
- Comparaison avec les branches *master* et *experimental* : `git diff master experimental`
- Le paramètre `[path]...` permet de filtrer la comparaison
- Il est possible d'ouvrir l'outil graphique configuré avec la commande `git difftool`

Les conflits

- Les opérations réalisées avec Git peuvent générer des conflits
- Les commandes `git statut` et `git diff` permettent d'identifier les conflits
- La gestion des conflits consiste à éditer le code en retirant les informations d'identification de conflit
- La syntaxe simplifiée d'identification de conflit permet d'utiliser des outils pour faciliter cette tâche
- La commande `git mergetool` ouvre l'outil graphique configuré

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> prob53:index.html
```

Exemple de conflit

Gestion des conflits

Notes

Git

Partage de travail et collaboration

Version 1.1

- Mise en place d'un dépôt distant
- Les branches distantes
- Récupération des modifications
- Publier ses modifications

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Mise en place d'un dépôt distant

Principes

- L'installation d'un dépôt sur un serveur consiste à y copier le répertoire Git et à configurer son accès
- Un dépôt serveur est dit "nu" ou *bare* (l'option `--bare` de `git clone` crée un dépôt "nu" à partir d'un autre)
- Les droits en lecture/écriture des utilisateurs sur le dépôt serveur conditionnent leur accès au dépôt
- Le processus d'installation d'un serveur Git peut rapidement devenir complexe

Solutions clés en main

- L'hébergement d'un dépôt serveur est nécessaire pour une utilisation collaborative de Git
- Différentes solutions d'hébergement de dépôts Git existent
 - Services : services en ligne proposant d'héberger des dépôts Git (Bitbucket, Github, Savannah, ...)
 - Auto-hébergement : applications à installer sur des serveurs personnels (GitLab, gitolite, gitweb, Gogs, ...)
- Il faut opter pour une solution adaptée aux utilisateurs et aux administrateurs

Public ou privé

- Tout dépend de la nature du dépôt et de son niveau de confidentialité
- Souvent les dépôts locaux sont privés (car inaccessibles) mais référencent un dépôt public (ou privé) partagé

Mise en place d'un dépôt distant

Notes

Certains groupes de développeurs utilisent Git à travers des *mailing lists* où ils s'échangent des patches et rendent leur dépôt accessible en lecture avec gitweb

Les branches distantes

- Une branche distante référence (pointeur) une branche d'un dépôt distant
- Les branches distantes ne peuvent pas être modifiées : elles représentent le travail d'autres collaborateurs
- Format du nom d'une branche distante :
remotes/<sourcename>/<branchname>
- Par défaut `git clone` nomme la source distante *origin* et copie la branche *master*
- Le fichier de configuration du dépôt `.git/config` contient des informations de correspondance avec le dépôt distant

```

steven@midgar:~/lab$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = https://github.com/enten/lab.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[branch "experimental"]
    remote = origin
    merge = refs/heads/experimental
  
```

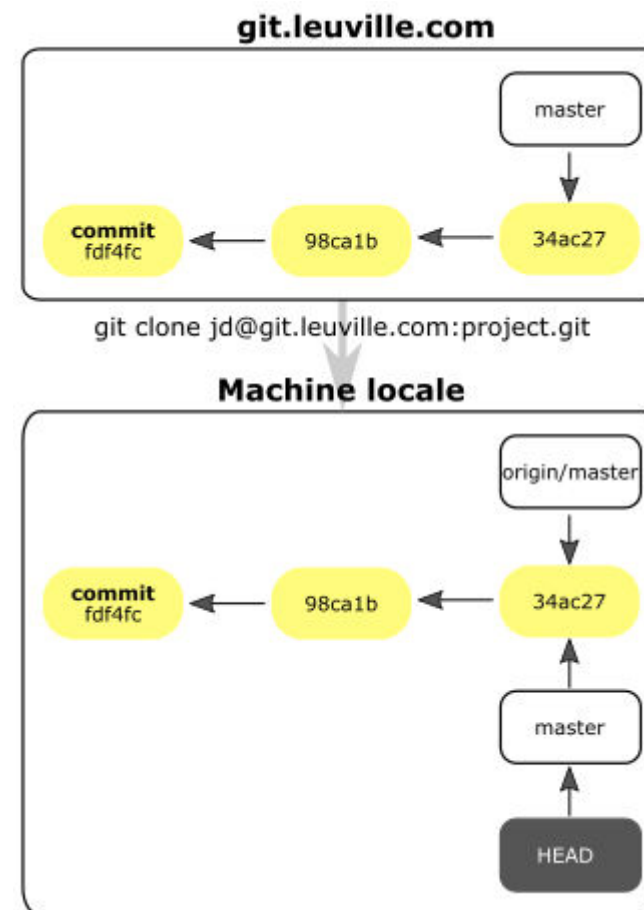


Illustration d'un clonage de dépôt

Les branches distantes

Notes

La commande `git remote` permet de gérer les dépôts distants
Sans arguments, la commande liste les dépôts distants

Rappel : la commande `git branch -r` liste les branches distantes

Récupération des modifications

- Sans contact avec le serveur du dépôt public les branches distantes n'avancent pas
- La récupération des modifications des branches distantes n'est pas automatique
- Il faut utiliser la commande `git fetch <sourcename>` pour récupérer les modifications distantes
- Si la branche locale et la branche distante ont avancées : il y a divergence
- Il faut alors fusionner la branche distante *origin/master* à la branche locale courante (*master*)
- De possibles conflits peuvent apparaître : il faut alors les corriger et faire un *commit* de cette opération
- La commande `git pull` permet de récupérer et de fusionner les modifications en une seule opération

Supprimer une branche distante :

```
git push origin --delete branchname
```

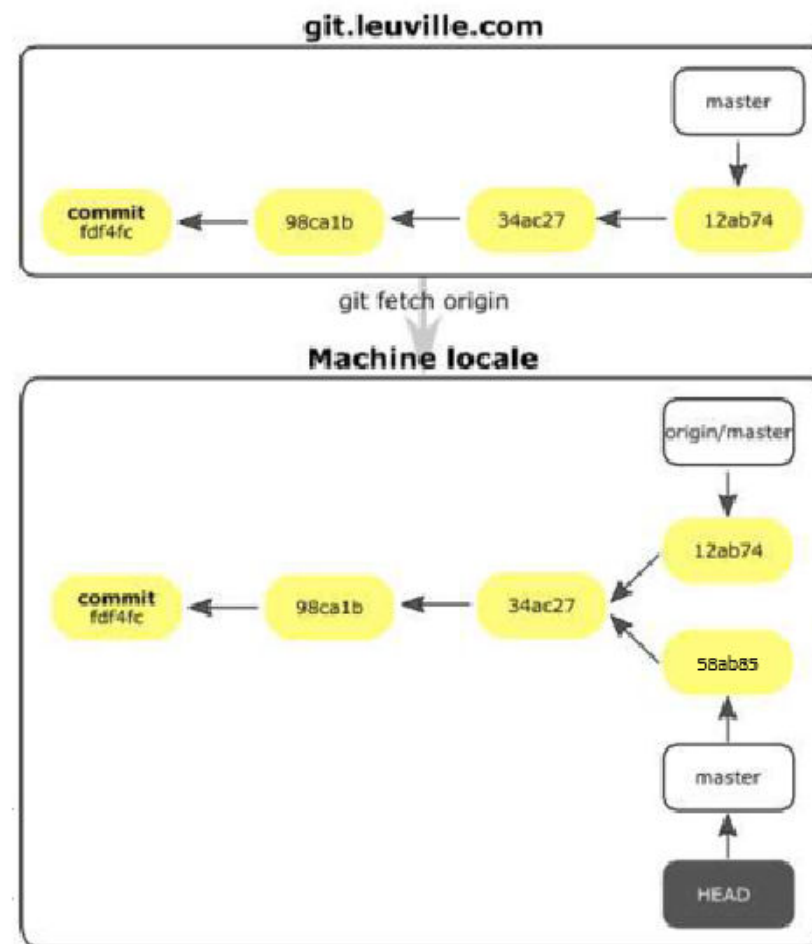


Illustration d'une divergence avec un dépôt distant

Récupération des modifications

Notes

Publier ses modifications

- La mise à jour d'un dépôt distant consiste à "pousser" (*push*) les modifications locales
- Cette opération n'est pas permise si le dépôt locale n'est pas à jour
- Avant de pousser des modifications, il faut s'assurer d'avoir récupéré celles distantes
- La commande `git push` s'utilise de différentes manières
 - Sans arguments, c'est la branche courante qui est poussée vers le dépôt associé (notion de branche "suivie")
 - Il est également possible de passer le nom du dépôt distant (*origin* par exemple) et de la branche à pousser
- Des droits en écriture sont nécessaires pour pousser son travail
- Les clés SSH permettent d'éviter de saisir le mot de passe à chaque publication de modifications

L'option de configuration `credential.helper` avec la valeur `cache` permet de mémoriser le mot de passe pour pousser des modifications vers un serveur identifiant les utilisateurs par *login/password*

Publier ses modifications

Notes

GIT

Mise en oeuvre des outils Git

Version 1.1

- Git-Gui/Gitk et SourceTree : clients graphiques Git pour Windows
- TortoiseGit : l'extension Git pour l'explorateur Windows
- GitWeb : l'interface Web de navigation au sein de dépôts Git
- GitHub : service Web d'hébergement de dépôts Git
- GitLab et Gogs : alternatives auto-hébergées à GitHub
- Gerrit : application de revue de code

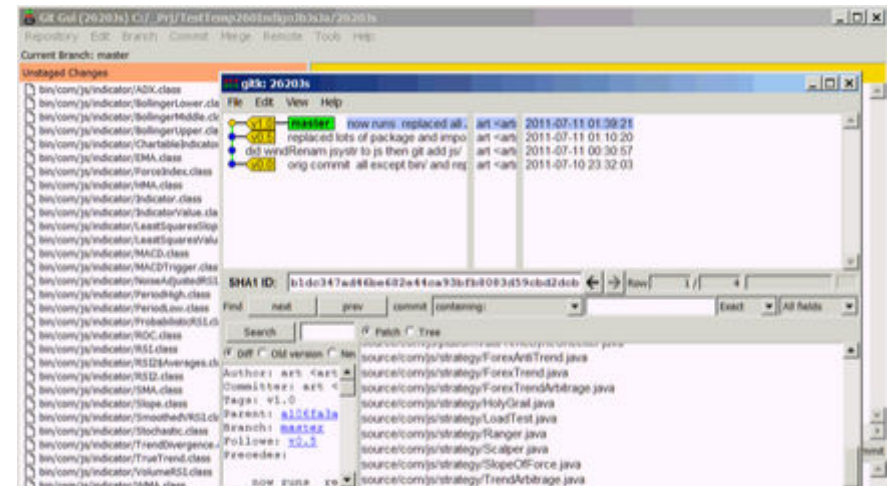
(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Git-Gui/Gitk et SourceTree : clients graphiques Git pour Windows

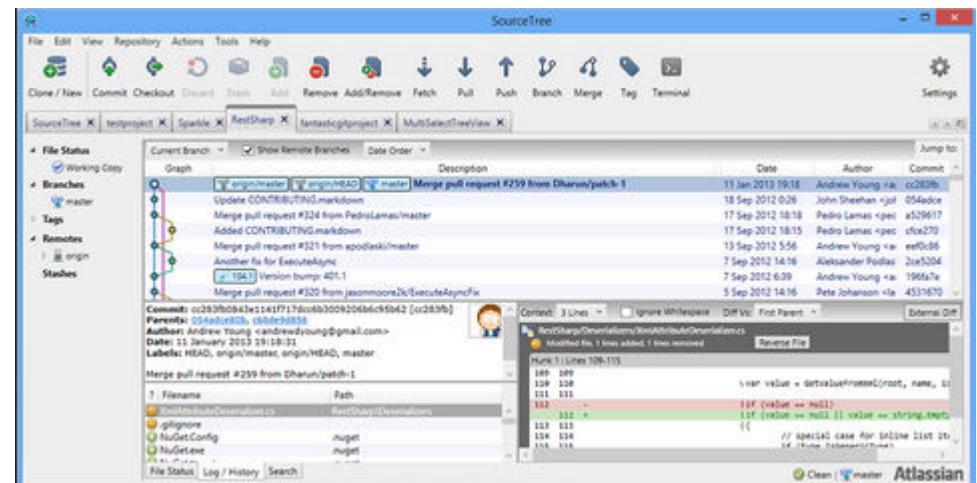
Git-Gui

- Interface graphique portable pour Git
- C'est un projet indépendant de Git :
<http://repo.or.cz/w/git-gui.git/>
- Il peut être installé depuis le gestionnaire de packages (avec *msysgit* pour Windows) ou par compilation
- Git-Gui se conjugue avec Gitk : le premier pour les fonctionnalités de *commits* et l'autre pour l'historique



SourceTree

- Client graphique Git/Mercurial pour Windows et Mac
- SourceTree propose des fonctionnalités pour gérer tous les aspects d'un dépôt
- Il est beaucoup plus complet que *Git-Gui* et n'a pas besoin d'outils complémentaires (*msysgit* ou *gitk*)
- Développé par Atlassian l'éditeur de Bitbucket (service d'hébergement de dépôts Git et Mercurial)
- Site Internet : <http://www.sourcetreeapp.com/>



Git-Gui et SourceTree : clients graphiques Git pour Windows

Notes

D'autres clients graphiques existent :

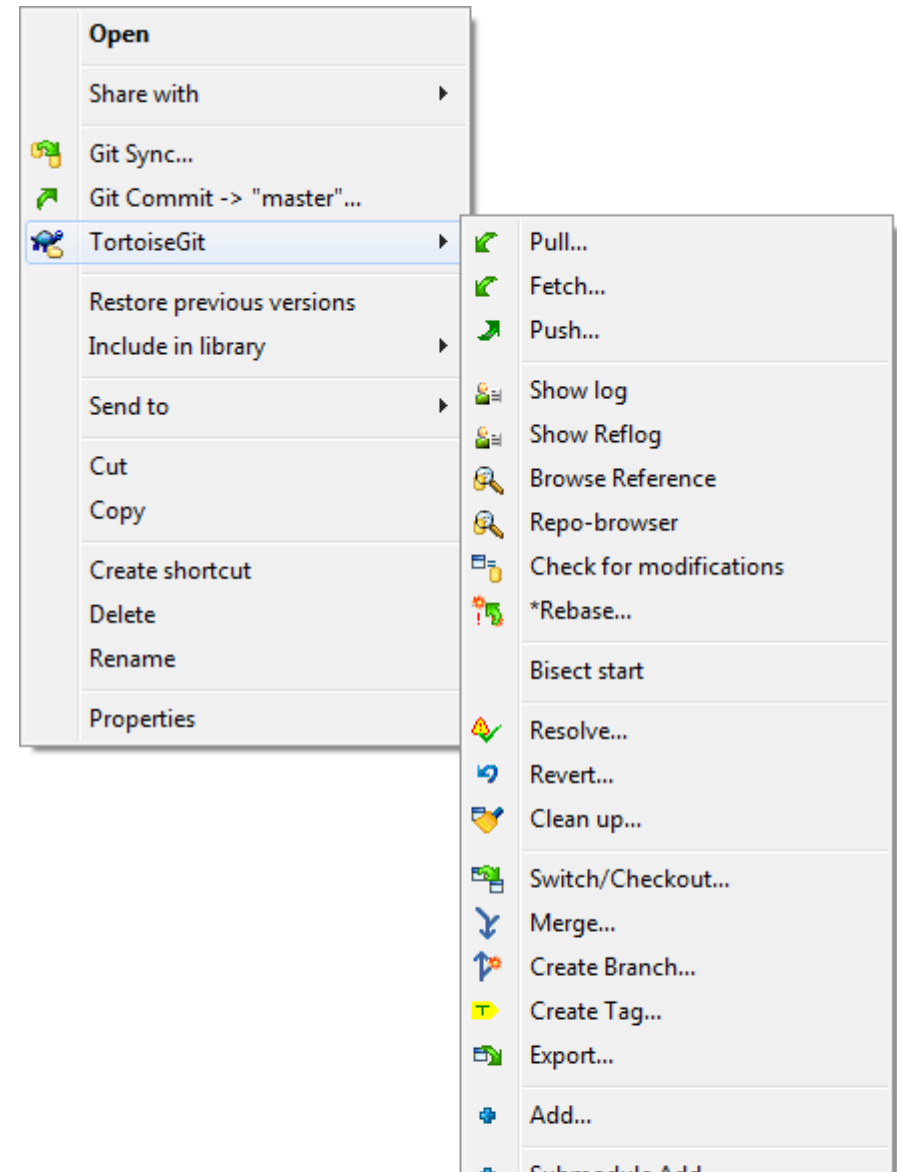
- Git Cola : <https://git-cola.github.io/>
- GitG : <https://wiki.gnome.org/Apps/Gitg>
- QGit : <http://sourceforge.net/projects/qgit/>
- ...

La documentation officielle liste plusieurs clients graphiques :

<http://git-scm.com/downloads/guis>

TortoiseGit : l'extension Git pour l'explorateur Windows

- TortoiseGit permet de gérer de manière graphique des dépôts Git
- Il étend l'explorateur Windows en intégrant des actions dans le menu contextuel
- Il nécessite l'installation de *msysgit* pour fonctionner
- Open source : <https://tortoisegit.org>
- Tortoise est également disponible pour d'autres VCS (CVS, SVN, Mercurial et Bazaar)



TortoiseGit : l'extension Git pour l'explorateur Windows

Notes

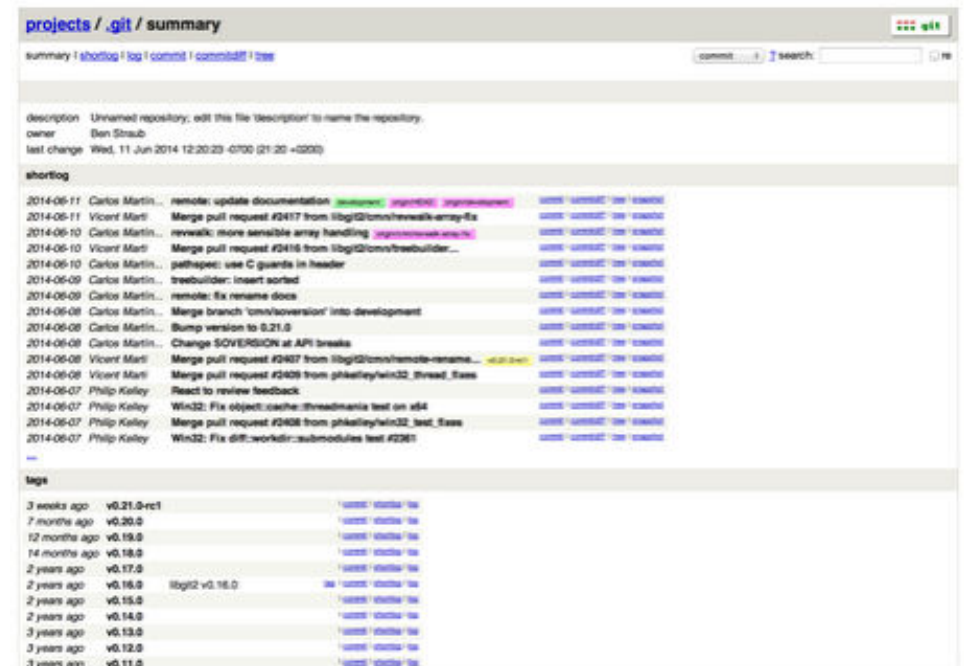
GitWeb : l'interface Web de navigation au sein de dépôts Git

Présentation

- GitWeb est une interface Web de visualisation de dépôts Git simplifiée
- Il s'agit d'un script CGI fournit par Git
- Le script nécessite l'usage d'un serveur Web (lighttpd, webrick, apache2, etc)

Installation

- Instance de serveur temporaire : utiliser la commande `git instaweb [--httpd=servername]` depuis un dépôt à visualiser (le script écoute sur le port 1234)
- Instance permanente : il faut utiliser le script du dépôt Git et configurer un VirtualHost Apache2



GitWeb : l'interface Web de navigation au sein de dépôts Git

Notes

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT = "/opt/git" prefix = /usr gitweb
  SUBDIR gitweb
  SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
  GEN gitweb.cgi
  GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Compilation du script CGI

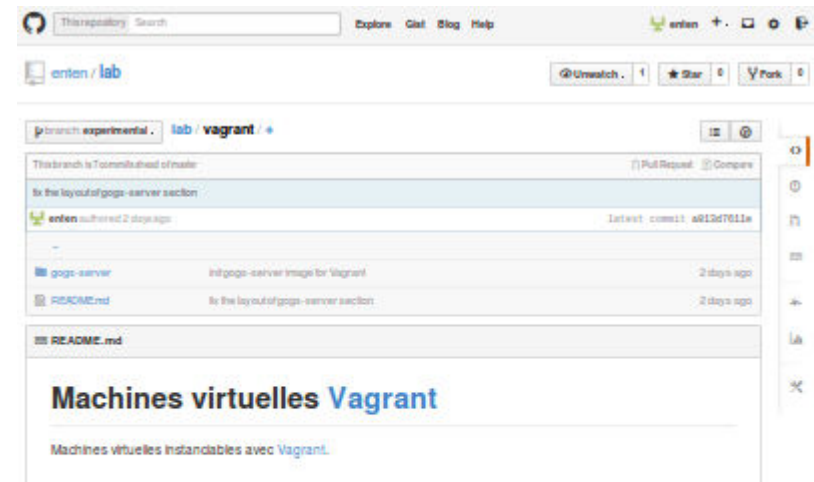
```
<VirtualHost *:80>
  ServerName gitserver
  DocumentRoot /var/www/gitweb
  <Directory /var/www/gitweb>
    Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
    AllowOverride All
    order allow,deny
    Allow from all
    AddHandler cgi-script cgi
    DirectoryIndex gitweb.cgi
  </Directory>
</VirtualHost>
```

Nouveau VirtualHost dans Apache pour utiliser le script CGI

Certaines distributions Linux (comme Ubuntu) propose un package gitweb

GitHub : service Web d'hébergement de dépôts Git

- Propose d'héberger des dépôts Git en ligne
- Il est exploité par l'entreprise GitHub Inc. qui propose des comptes professionnels payants
- L'hébergement est gratuit pour des dépôts publics et payant pour des dépôts privés
- Des fonctionnalités de Wiki et de suivi de problèmes sont disponibles
- GitHub introduit une fonctionnalité de *fork*
 - Un fork consiste à cloner un dépôt public pour y apporter des modifications
 - Il faut ensuite soumettre une *pull request* à l'équipe du projet initiale pour faire intégrer des *commits*
- Ce mode de fonctionnement fait débat au sein de la communauté Open Source
- D'autres services complémentaires sont proposés : Gist et Travis CI



Bitbucket est un service Web concurrent de GitHub

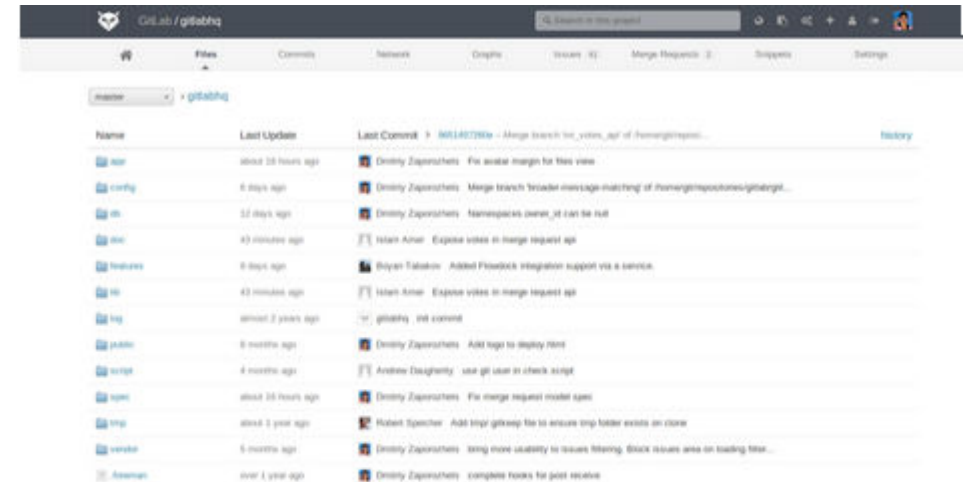
GitHub : service Web d'hébergement de dépôts Git

Notes

GitLab et Gogs : alternatives auto-hébergées à GitHub

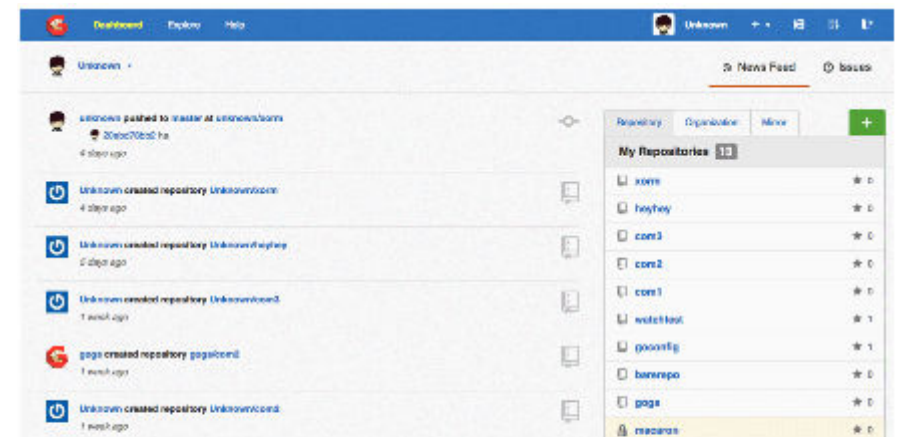
GitLab

- o Au départ une alternative libre auto-hébergée à GitHub
- o GitLab s'est développé et propose plusieurs offres
 - o GitLab Community Edition (CE) : application libre pour auto-héberger ses dépôts
 - o GitLab Enterprise Edition (EE) : édition payante et propriétaire proposant plus de fonctionnalités
 - o GitLab.com : service en ligne d'hébergement gratuit de dépôts publics ou privés



Gogs

- Présenté comme une alternative en Go à GitLab CE
- Gogs est Open Source et profite des avantages de Go
 - Installation simplifiée via un binaire unique (le déploiement de GitLab est plus complexe)
 - Multi-plateformes : Windows, Mac, Linux, ARM, etc
- L'exécution de Gogs nécessite moins de ressources que GitLab (1GB de RAM minimum à l'exécution pour GitLab contre 50MB pour Gogs)



GitLab et Gogs : alternatives auto-hébergées à GitHub

Notes

Gerrit : application de revue de code

- Application Web libre de revue de code de dépôts Git
- Il permet à ses utilisateurs d'approuver ou de rejeter les modifications du code source
- Outils développé par Google et utilisé pour le développement d'Android
<https://android-review.googlesource.com>
- L'objectif est de centraliser d'avantage le travail des développeurs sur de gros projets
- Gerrit est codé en Java (depuis la version 2) et nécessite l'utilisation d'une base SQL
- Quelques liens utiles

0 Site officiel : <https://code.google.com/p/gerrit/>

- Documentation :
<https://gerrit-documentation.storage.googleapis.com/Documentation/2.11/index.html>

- o **Guide d'installation :**
<https://gerrit-documentation.storage.googleapis.com/Documentation/2.11/install.html>



Le service GerritHub permet de faire de la revue de code sur des dépôts GitHub

Gerrit : application de revue de code

Notes