Introduction à Git

Guillaume Allègre
Guillaume.Allegre@silecs.info

URFIST Lyon

2014

Licence Ceative Commons By - SA

- Vous êtes libre de
  - **partager** — reproduire, distribuer et communiquer l'oeuvre
  - **remixer** — adapter l'oeuvre
  - d'utiliser cette œuvre à des fins commerciales
- Selon les conditions suivantes
  - **Attribution** — Vous devez attribuer l'oeuvre de la manière indiquée par l'auteur de l'oeuvre ou le titulaire des droits (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'oeuvre).
  - **Partage à l'identique** — Si vous modifiez, transformez ou adaptez cette oeuvre, vous n'avez le droit de distribuer votre création que sous une licence identique ou similaire à celle-ci.

`http://creativecommons.org/licenses/by-sa/3.0/deed.fr`

Contribuer - Réutiliser

Ce document est rédigé en LaTeX + Beamer.

Vous êtes encouragés à réutiliser, reproduire et modifier ce document, sous les conditions de la licence *Creative Commons, Attribution, Share alike 3.0* précédemment décrite.

J'accepte volontiers les remarques, corrections et contributions à ce document.

Plan de la formation

- Les bases de Git
- Versionner ses fichiers
- Travailler à plusieurs
- Gérer des branches
- Déboguer avec git
- Introduction au *social coding*

Les bases de Git

Gestionnaire de versions

Points communs

- issu du développement : fichiers **textes** et binaires
- un projet : un ensemble de fichiers et répertoires, interdépendants
- archive les versions *successives* d'un projet
- stocke les versions *parallèles* d'un projet (branches)
- facilite le travail collaboratif *asynchrone*
- interface en ligne de commande ; interface graphique optionnelle

Différenciations

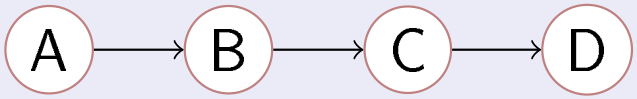
- centralisé (CVCS) ou distribué (DVCS)
- avec ou sans verrou, validation, processus d'intégration...
- contexte : outil d'entreprise vs. outil communautaire

Bref historique des gestionnaires libres

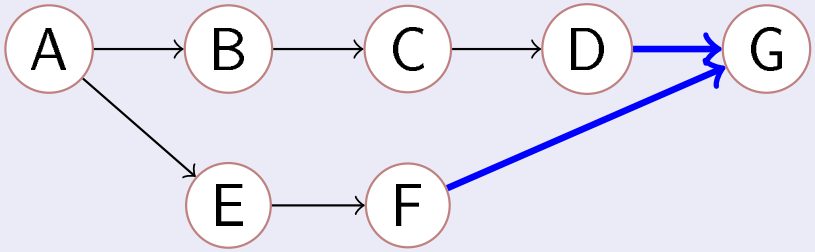
- Gestionnaires locaux
  - RCS (revision control system) 1982-
- Gestionnaires centralisés
  - CVS (*concurrent versionning system*) 1990-2008
  - SVN (Subversion) 2000-
  - en perte de vitesse mais encore actifs
- Gestionnaires distribués (2000's)
  - ancêtres : Gnu ARCH (2001-2006), Darcs (2002-)
  - GNU Bazaar (Canonical LTD, 2005-2014 ?)
  - **Mercurial** (hg) :
  - **Git** : issu du noyau Linux

Une idée rapide de Git

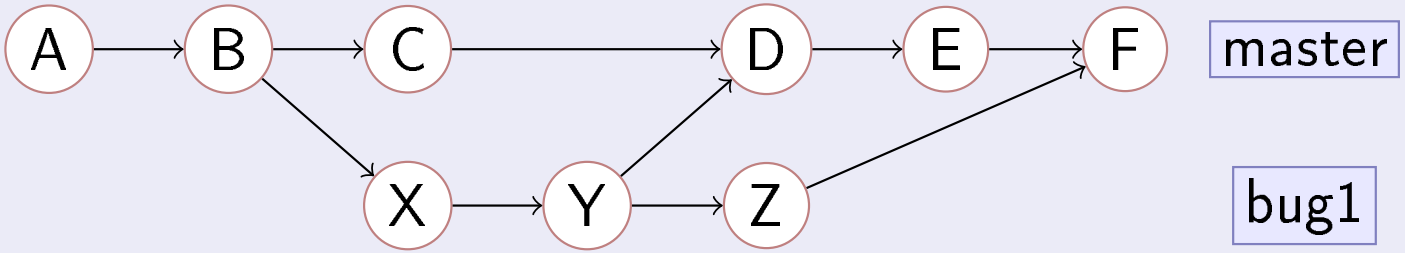
Historique simple



Fusion simple



Fusions complexes



Git en pratique

Environnement Unix

- commande `git`
- aide : `git help` + `git help <command>` ou `man git-<command>`
- `gitk` interface graphique minimale

Environnement Windows

- mini-environnement unix `MinGW`
- tout ce qui précède
- (optionnellement) intégration Explorateur

Intégration extérieure

- IDE environnements de développement intégrés
- gestionnaire de bugs, de tâches...
- *social coding*

Premier contact - TP

1. `git clone https://github.com/demosilecs/formagit-webp`
   ou `git clone`
   `git@git.silecs.info:temporaire/formagit-webp`
2. Combien y a-t-il de fichiers dans le projet ?
3. Combien y a-t-il eu de contributions successives (*commits*) ?
4. Par combien de contributeurs ?
5. Combien y avait-il de fichiers dans la première version du projet ?
6. Qui a introduit le fichier d'image ?
7. Que fait Carl dans le projet ?

8. Répondre aux mêmes questions en ligne de commande.

Premiers concepts

- **Commit**
- **Message de commit**
- **Hash (SHA-1)**
- **Patch** ou **Diff**
  référence aux commandes shell `diff` et `patch`
- **Auteur (utilisateur)**
- **Branche (master par défaut)**
- **Dépôt git** (*repository*)

Versionner ses fichiers localement

TP - premières modifications locales

1. Configurer son identité : `git config user.name "Prenom Nom"` et `git config user.email qui@quoi`
2. Ajouter un paragraphe (ou plus...) dans le fichier principal.
3. Vérifier les changements (`git status`, `git diff`) puis commiter (`git add`, `git commit`).
4. Répéter l'opération, en modifiant plusieurs fichiers (*ad lib.*)
5. Naviguer dans l'historique avec `git checkout <hash>` (en arrière)
6. Que se passe-t-il si on essaie de commiter maintenant ?
7. Repentir profond : comment annuler un changement fait à la question 2-3 ? ...

Les différents statuts d'un fichier

- Trois "espaces" différents
  - le répertoire de travail (système de fichiers standard) (*working dir*)
  - l'*index* (aka *cache* aka staging area)
  - le dépôt local (*repository*)

- Travaux pratiques
  1. utiliser `git status` pour suivre les statuts d'un fichier
  2. sur un graphe simple, représenter les transitions fournies par les commandes entre les 4 états :

non suivi	suivi		
	non modifié	modifié	"indexé" (staging area)

Méthodologie : comment commiter ?

- Des messages de commit explicites (éventuellement règles de projet)

- Commits atomiques
  Deux changements indépendants *devraient* être commités séparément

- Commits cohérents
  Tous les changements liés entre eux *devraient* être commités ensemble

- En cas d'erreur
  - il est possible de regrouper des commits séparés (difficulté moyenne)
  - il est possible de scinder un commit (difficulté élevée)
  - cela reste des opérations sensibles (...)

TP - Créer son propre dépôt

1. Hors de l'espace géré par git, créer un nouveau répertoire (dossier).
2. `git init` initialise un dépôt vide (quel que soit l'état du répertoire).
3. Créer ou copier quelques fichiers et reprendre l'expérimentation du TP précédent.
4. Modifier un fichier. Repentir : annuler les modifications locales avec `git checkout -- <fichier>`.
5. Supprimer un fichier localement (`rm`). Comment le restaurer ?
6. Supprimer un fichier localement, le restaurer, puis le supprimer *du dépôt* avec `git rm`.
7. Renommer un fichier localement. Que donne un `git status` ? Renommer dans le dépôt avec `git mv`.

Travailler à plusieurs

Les dépôts distants

- Organisation des dépôts distants
  - 0, 1 ou plusieurs dépôts distants
  - selon la forme du projet (local, centralisé, distribué...)
  - droits d'accès différenciés

- En pratique
  - `git remote [-v]` liste les dépôts distants
  - `git clone URL` initialise un dépôt local à partir d'un distant
  - `origin` le nom par défaut (source du clone)

Les protocoles et URL

- local (système de fichiers local)
  - `git clone /opt/git/project.git`
  - ou `git clone file:///opt/git/project.git`
  - authentification OS, accès en lecture + écriture
  - envisageable pour un système de fichier réseau (NFS...)
- ssh
  - `git clone ssh://user@server/project.git`
  - authentifié, accès en lecture + écriture
  - le plus répandu en contribution
- git
  - `git clone git://server/project.git`
  - aucune authentification, accès anonyme, lecture seule
- http(s)
  - `git clone https://server/project.git`
  - lourd, authentification web

TP - Premiers dépôts distants

1. Cloner un dépôt local
2. Vérifier les serveurs distants
3. Ajouter le dépôt distant "d'origine" sous un alias `<lequel>`
4. Vérifier les informations dessus : `git remote show <lequel>`

5. `git fetch <lequel>`

6. Accessoirement, redonder le dépôt distant et utiliser `git remote rename <old> <new>` et `git remote rm <new>`

Concepts - dépôts et branches (intro)

- `git fetch` met à jour le cache des branches distantes.
  Ne modifie aucun fichier local.
- `git pull` met à jour le cache des branches distantes ET fusionne dans la (les) branche(s) locale(s) correspondante(s).
  Modifie les fichiers locaux.
- `git push [origin] [master]` propage les commits locaux dans la branche cible.

TP - contributions collaboratives (expérimentation)

1. Reprendre un clone neuf de `webproject`
2. Ajouter des contenus (textes et fichiers)
3. Les pousser dans `origin` et récupérer les modifications des autres développeurs.

Gérer des branches

Gérer les tags

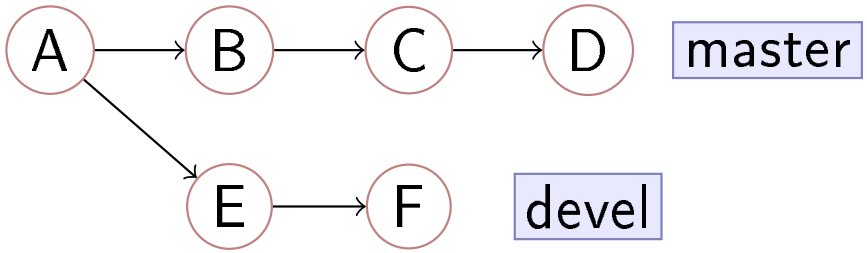
- Un tag = un alias sur un commit (branche fixe)
  - utilisé pour gérer les diffusions (*releases*) du projets
  - optionnellement, pour approuver et signer (GPG) un snapshot
- Lister et consulter les tags
  - `git tag`
  - `git tag -l <motif>`
  - `git show <mon-tag>`
- Ajouter un tag
  - tag annoté : `git tag -a <mon-tag> [-m "mon message"]`
  - tag léger : `git tag <mon-tag>`
- Partager un tag
  - `git push <origin> [mon-tag]`

Les branches

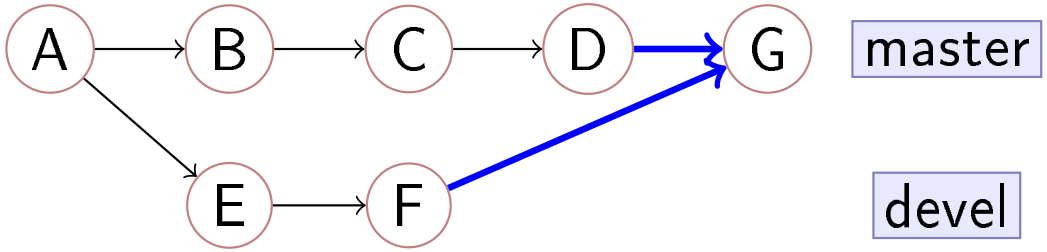
- Cadre théorique
  - stockage git : une série de snapshots (cf. suivant)
  - parents d'un commit : 0 (initial), 1 (cas général), plusieurs (fusion)
  - une branche : un pointeur mobile vers un commit
  - `master` : branche par défaut, créée au premier commit
  - `HEAD` nom symbolique du pointeur courant
- En pratique : créer une branche `<testing>`
  - *optionnellement* `git checkout <hash>`
  - `git branch <testing>`
  - *optionnellement* `git checkout <testing>`
  - ou `git checkout -b <testing>` : condense les deux précédentes

Fusionner : `git merge`

- situation d'origine

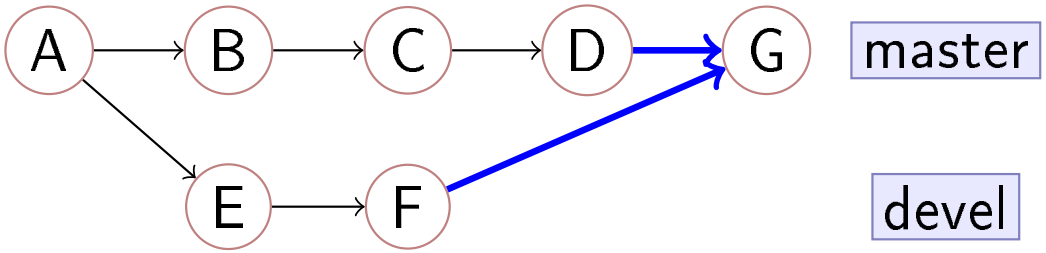


- `git checkout master`
- `git merge devel`
- fusion 3-way merge (recursive)



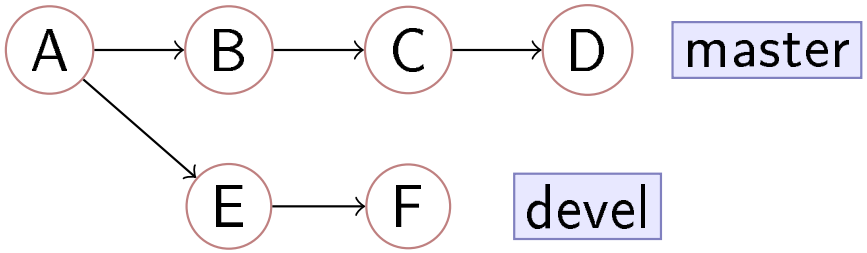
Fusionner : `git merge` en conflit !

- fusion 3-way merge (recursive)
  - `A` est l'ancêtre commun
  - `D` est le parent 1 (branche master)
  - `F` est le parent 2 (branche devel)

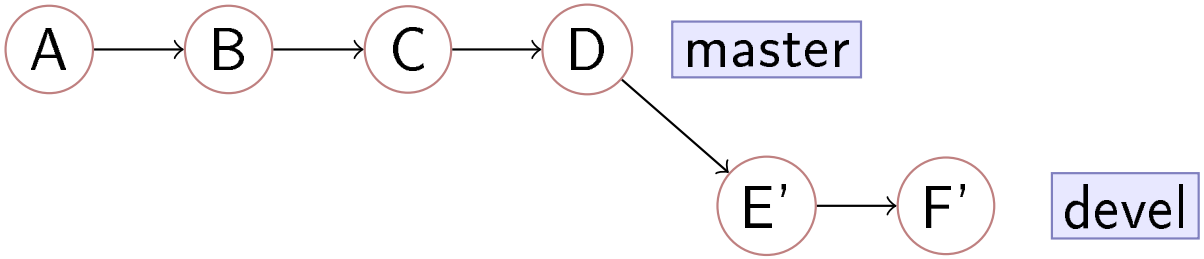


- Gestion des conflits
  - `git mergetool` éditeur "graphique"
  - `git merge --abort` abandonner la fusion ; retour à l'état antérieur
  - `git add <file>` marquer le conflit résolu

Rebaser : `git rebase`



- situation d'origine
- `git checkout devel`
- `git rebase master`

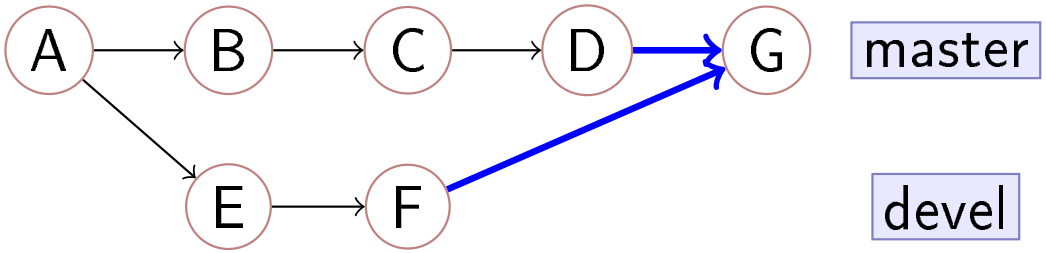


- après rebase

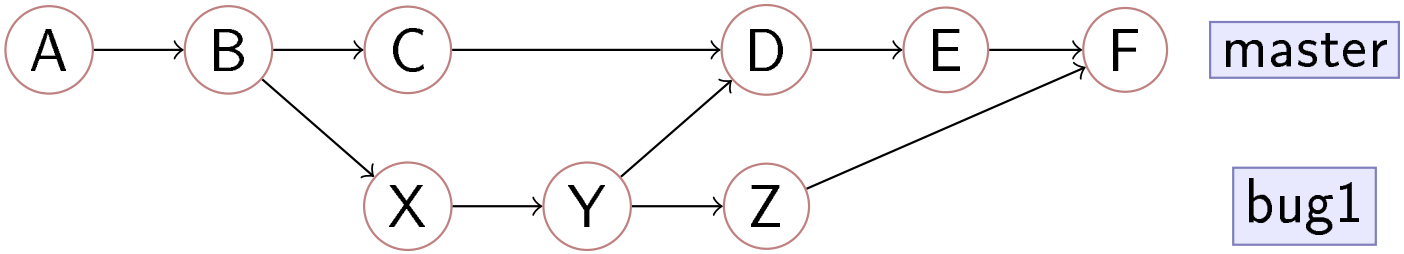
TP Manipulation de branches

À partir d'un dépôt quelconque, réaliser la séquence de commits correspondant aux graphes suivants :

1. Fusion simple



2. Fusions complexes



Outils de gestion des branches

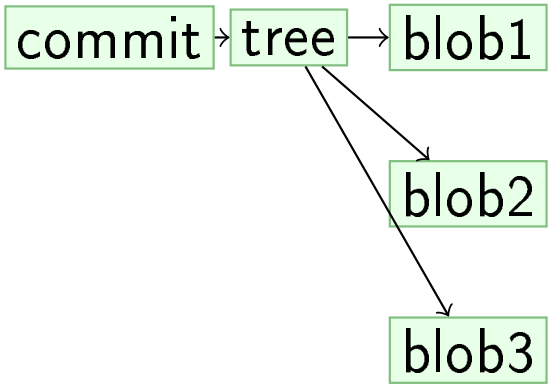
- Exploration des branches
  - `git branch -v` affiche les détails (commit)
  - `git branch -a` affiche les branches distantes
  - `git branch --merged` affiche les branches fusionnées dans la courante
  - `git branch --nomerged` l'inverse !
- Élagage
  - `git branch -d ou --delete` supprime une branche (déjà fusionnée)
  - `git branch -D` supprime une branche (inconditionnellement)

Branches distantes

- commande `git branch -a`
- Une branche locale peut suivre (*track*) une branche distante.
  ex. `git checkout -b devel --track origin/devel`
- TP - éditions collaboratives et fusions
  1. Pourquoi le push échoue (commit refusé) si d'autres ont déjà modifié dans l'intervalle ?
  2. Utiliser un `git pull` puis retenter le `git push`
  3. Pourquoi le `pull` a-t-il résolu la situation ? Que fait-il en interne ?
  4. Symétriquement, il exite aussi `git pull --rebase`. Même question que précédemment.
  5. Quel est l'intérêt d'utiliser `git pull` plutôt que `git pull --rebase` ?

Pour aller plus loin... stockage dans Git

- commit : métadonnées, hash et pointeur sur un arbre
- arbre : pointeurs sur des blobs
- blob : une révision d'un fichier



Outils Git

TP - Défaire ce qui a été fait

1. Oublier les modifications locales d'un fichier
   - `git checkout -- <fichier>`
   - Danger écrasement du fichier
2. "Désindexer" un fichier
   - faire un `git add` puis un `git status` sur un fichier modifié
   - `git reset [--mixed] HEAD <fichier>`
3. Annuler un ancien commit

   `git revert [--no-commit] <commit-hash>`

Déboguer avec Git

- Qui blâmer (et pourquoi) ?
  - `git blame [-n] <fichier>` préfixe chaque ligne
  - `git show <hash>` restitue le contexte du commit
- Recherche dichotomique
  - `git bisect start`
  - `git bisect bad` sur le commit courant
  - `git bisect good <last-good-hash>`
  - réitérer `git bisect <good|bad>` autant que nécessaire
  - `git bisect reset`
  - automatisation possible : `git bisect run checkgood.sh`
    où `checkgood.sh` retourne 0 si projet OK, autre valeur sinon

Rechercher dans le dépôt

- Chercher dans les métadonnées de commit
  - ex. `git log --grep=<motif>`
  - ex. `git log --author=<motif>`

- Chercher dans les fichiers du dépôt
  - ex. `git grep [-l] <motif>`
  - ex. `git grep [-l] <motif> <hash>`

Réécrire l'historique

- `git commit --amend` (niveau simple)
  - `git commit -m 'commit normal'`
  - optionnellement `git add fichier-oublié` (ou `edit` + `git add`)
  - `git commit --amend`

- `git rebase --interactive` en commits locaux (niveau moyen)
  - ex. `git rebase -i <base-hash>` passe en mode interactif
    - `pick` utilise le commit
    - `reword` utilise le commit en modifiant le message
    - `edit` utilise le commit, interruption interactive pour modif
    - `squash` utilise le commit, mais fondu dans le précédent
    - `fixup` comme "squash", en oubliant le message de commit

- idem sur commits publiés fortement déconseillé

Stashing

- Scénario d'utilisation
  - On travaille sur de grosses modifications (état "sale")
  - On doit d'urgence changer de branche (bugfix...)
  - On ne veut pas commiter l'état en cours
- En pratique (TP)
  1. Modifier quelques fichiers courants puis `git status`
  2. `git stash` pour ranger les modifications dans la "cachette" puis `git status`
  3. `git stash list` : il peut y avoir plusieurs cachettes !
  4. `git checkout ...` : on change de contexte
  5. `git stash pop`
  6. ou éventuellement `git stash apply [stash@{0}]` ...

Autres outils et techniques Git

- `git cherry-pick <hash>` pour appliquer un commit distant sans fusion
- Signature des objets Git (PGP / GPG)
  - `git commit -S` signe un commit
  - `git tag -s` signe un tag annoté
- Configurations
  - `git config [--global]` et `./git/config`
  - `.gitignore`

Introduction au *social coding*

Modèles de branches

- Idée
  - une ou deux branches "longues", master ou production
  - quelques branches longues fusionnées périodiquement, ex. développement
  - des branches courtes (fonctionnalités), fusionnées puis détruites
- Une proposition de modèle
  - branche `master`
  - branche `production` (ou `releases`)
  - branche `testing`
  - branche `devel`
  - branches "fonctionnalités" (*feature branch*)
  - branche `bugfixes`
- Quelques références
  - nvie.com/posts/a-successful-git-branching-model/
  - git-scm.com/book/en/Git-Branching-Branching-Workflows

GitHub

- GitHub `http://www.github.com`
  - Société *GitHub, Inc.* (US, 2008)
  - modèle économique : projet ouvert gratuit, projet privé payant
  - environ 13M dépôts Git
- Fonctionnalités Git de base
  - créer des projets Git
  - enregistrer des utilisateurs (authentification ssh)
- Fonctionnalités additionnelles
  - enregistrement d'*organisations* et d'*équipes*
  - un bugtracker pour chaque dépôt
  - flux, abonnements, wikis... sur les objets du dépôt
  - statistiques paramétrables sur chaque dépôt
  - éditeur en ligne (navigateur web)
  - accès aux *pull requests* : contribution *upstream*
  - profils développeurs ($\longrightarrow$ service recrutement)

Alternatives à GitHub

- Gitorious `http://www.gitorious.org`
  - Lancé en janvier 2008
  - Fourni par Gitorious AS (NO), puis Powow AS (NO-PL)
  - Code de la plateforme libre depuis l'origine (AGPLv3)

- Bitbucket `http://www.bitbucket.org`
  - Lancé en 2008, basé sur Mercurial
  - Depuis 2010, fourni par *Atlassian* (AU), éditeur de JIRA
  - Accepte Git depuis octobre 2011