Introduction à Git

François Gannaz <francois.gannaz@silecs.info>

Silecs

Plan de la formation

Git : suivi de versions décentralisé

suivi de versions

- gère un ensemble de fichiers de tous types
- enregistre à la demande l'état de tous ces fichiers
- conserve l'historique (version successives)
- peut gérer des variantes (versions parallèles)

décentralisé

- chaque espace de travail est complet et indépendant
- il peut échanger avec d'autres, à la demande

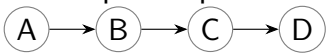
Interfaces de Git

- la ligne de commande
- éventuellement, interfaces graphiques
- éventuellement, intégration à l'IDE, etc

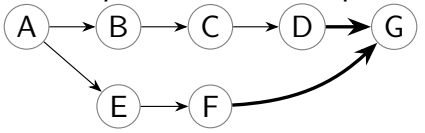
Une idée rapide de Git en graphes

Chaque lettre ci-dessous est un *point de sauvegarde* (commit) de l'ensemble des fichiers.

Historique simple



Historique avec versions parallèles



tronc principal

variante experimentale

Commandes et interfaces

Ligne de commande

- en shell : `git`
- commande git, par exemple `git help`
- options et paramètres, par exemple `git help -a`

Sous Windows, l'installateur contient un terminal et le shell Bash.

Autres interfaces

- graphiques, par exemple *gitk* (officiel)
- intégrées à l'IDE, par exemple Netbeans ou Eclipse
- intégrées au navigateur de fichiers

TP – Visualiser et naviguer

1. `git clone` https://github.com/silecs-demo/git-2017
2. Visualiser le projet dans un navigateur web.
3. Combien y a-t-il de fichiers dans le projet ?
4. Combien y a-t-il eu de contributions successives (*commits*) ?
5. Par combien de contributeurs ?
6. Combien y avait-il de fichiers dans la première version du projet ?
7. Qui a introduit le fichier d'image ? Dans quel commit ?
8. Que fait Carl dans le projet ?
9. Consulter dans son navigateur web les différentes modifications du projet depuis l'origine.
10. Répondre aux mêmes questions en ligne de commande, en utilisant l'aide-mémoire.

Concepts et jargon de Git

dépôt	(repository) tout ce que gère Git pour un projet
copie de travail	(working tree) fichiers réels, placés dans le répertoire du projet
commit	enregistrement de l'état des fichiers du dépôt, à un instant donné
hash	identifiant du commit, calculé par SHA-1 et écrit en hexadécimal
révision	élément qui pointe vers un commit : *hash*, *branche* ou *tag* (cf loin)
diff, patch	différence entre deux groupes de fichiers, généralement de texte

Versionner en local

Le cycle local

1. Modifier ou créer des fichiers
2. `git status` : vérifier les fichiers impactés
3. `git diff` : vérifier les changements
4. `git add [fichiers]` : ajouter les changements
5. `git commit` : enregistrer ces changements
6. Revenir à l'étape [1-4]

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 dans le fichier principal.
3. Vérifier les changements, puis commiter.
4. Ajouter deux nouveaux fichiers au dépôt.
5. Modifier trois fichiers, mais n'enregistrer dans le dépôt qu'une seule modification.
6. Naviguer dans l'historique pour revenir à l'état d'avant ces modifications.
7. Que se passe-t-il si on essaie de commiter maintenant ?
8. Bonus : comment annuler le commit de la question 3 ?

Méthodologie des commits ?

Message d'un commit

- conventions selon le projet
- format recommandé :

```
ajoute un formulaire de contact (titre)

Sur la page /contact…
(description multi-lignes)
```

Contenu d'un commit

Pour garder un historique clair et réversible :

atomique Un commit ne fait qu'une chose.

cohérent Un commit fait cette chose complètement.

Plus loin, techniques pour regrouper, séparer, nettoyer (`rebase`…)

Comment comprendre le statut des fichiers ?

3 espaces distincts

- les fichiers (working tree)
- l'index, appelé aussi cache ou staging area
- le dépôt et ses commits

Commiter $\iff$ Enregistrer l'index dans le dépôt

Travaux Pratiques

1. Modifier un fichier, et commenter `git status` avant et après `git add`.
2. Modifier ce même fichier et commenter `git status`.
3. Tracer un graphe des transitions entre les 4 états pour un fichier

non suivi	*suivi*		
	non modifié	modifié	indexé

Les transitions sont associées à des commandes `git`.

TP — Créer un dépôt, autres opérations simples

1. Hors de l'espace géré par git, créer un nouveau répertoire.
2. `git init` initialise un dépôt vide dans le répertoire.
3. Créer ou copier quelques fichiers et les enregistrer dans le dépôt.
4. Modifier un fichier. L'ajouter à l'index. Modifier encore, puis annuler ces dernières modifications.
5. Annuler l'ajout, sans modifier le fichier.
6. Annuler la modification locale du fichier (revenir à l'état du dernier commit).
7. Supprimer un fichier *localement* (`rm`). Comment le restaurer ?
8. Supprimer un fichier *du dépôt* avec `git rm`.
9. Renommer un fichier localement. Que donne un `git status` ? Renommer dans le dépôt avec `git mv`.
10. Quel est l'impact d'un fichier `.gitignore` contenant une ligne `/*.gz` ?

Branches et fusions

Tags – références fixes

tag = nom donné à un commit, référence fixe

Opérations sur les tags

- Lister les tags : `git tag`
- Créer un tag : `git tag mon-tag [révision]`

Partout où un hash est utilisé, un tag peut le remplacer :

- `git show montag`
- `git diff montag`
- `git log montag1`

TP – Tags et révisions

1. Donner le nom *v1.0-alpha* au commit actuel.
2. Vérifier qu'il apparaît dans la liste des tags.
3. Attribuer le tag *start-point* au commit d'avant les modifications locales. Se placer en *start-point*.
4. Quel est l'historique entre ces deux repères ?
   Quels sont les changements ?
5. Que contient le commit qui crée la version 1.0 alpha ?
6. Ajouter les lignes suivantes à `~/.gitconfig` :
   ```
   [alias]
   log --pretty=format:\"%h %ad [%an] %d %s\" --graph --date=short
   ```
   Que donne `git hist` ?
7. En s'aidant de `git help revisions`, expliquer `git hist HEAD^2..v1.0-alpha`.

Branches – référence mobile

branche = nom donné à un commit, référence mobile
qui se déplace à chaque commit suivant.

Exemple avec master

- *master* : branche par défaut à la création d'un dépôt
- `git show master` : montre le dernier commit
- Chaque commit déplace le curseur de la branche active.

Opérations sur les branches

- Lister : `git branch`
  L'astérisque indique la branche active.
- Créer et activer : `git checkout -b mabranche`
- Créer sans activer : `git branch mabranche [rév]`
- Activer : `git branch ma-branche-existante`

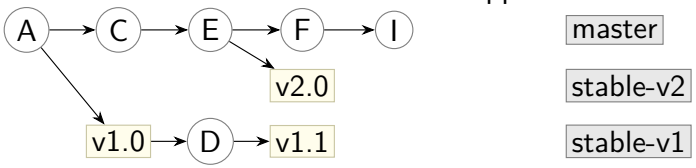
TP – Branches

1. Lister les 2 derniers commits avec `log` puis `hist`.
2. Depuis master, se placer dans une branche nommée *new-feature*, et consulter à nouveau l'historique.
3. Ajouter un nouveau fichier, commiter, et consulter à nouveau l'historique.
4. Utiliser `git branch -v`, puis se placer sur la branche *master*.
5. Modifier un fichier et le commiter.
6. Visualiser l'historique du dépôt local, avec `gitg` puis en ligne de commande avec l'option `--all` de `git log`.
7. Comment comprendre la commande suivante ?
   `git log master..new-feature`

Méthodologie des branches

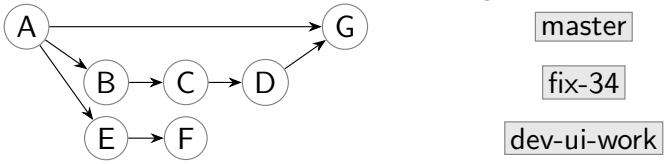
Pourquoi des branches **longues** ?

Branches de stabilisation et de développement :



Pourquoi des branches **courtes** ?

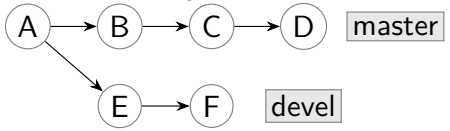
feature-branches et corrections de bugs :



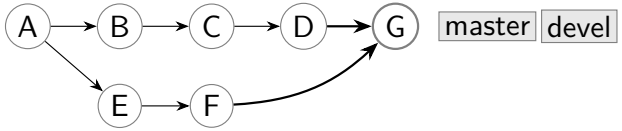
Fusionner des branches – merge

fusionner X dans Y = ajouter dans Y tous les changements de X
(changements depuis leur ancêtre commun)

- situation d'origine



- `git checkout master`
- `git merge devel`
- situation après fusion



Conflits à la fusion

Si la même zone a été modifiée *différemment* dans les deux branches, Git ne peut pas fusionner automatiquement.

En cas de conflit, deux options…

- Annuler la fusion : `git merge --abort`
- Résoudre ce conflit :
  1. Corriger un fichier en conflit
  2. `git add` sur ce fichier corrigé
  3. Répéter 1-2 tant qu'il reste des conflits
  4. `git merge --continue`

  `git mergetool` automatise les étapes 1-2 avec un outil externe (kdiff3, …)

Par défaut, les passages en conflit sont balisés par `<<< === >>>`
Conseil : `git config --add merge.conflictStyle diff3`

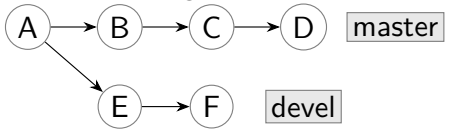
TP – Fusion de branches

1. Fusionner la branche *new-feature* dans *master*.
2. À quels commits correspondent ensuite ces deux branches ?
3. Supprimer la branch *new-feature* qui a été fusionnée.
4. Créer une branche *conflict*.
5. Ajouter des commits conflictuels sur *conflict* et *master*.
6. Fusionner *conflict* dans *master*.
7. Dans la situation de branches longues *stable-v1* et *stable-v2*, on souhaite corriger un bug critique qui affecte toutes les versions.
   Comment créer la branche de correction ?
   Comment appliquer cette même correction aux 3 branches ?

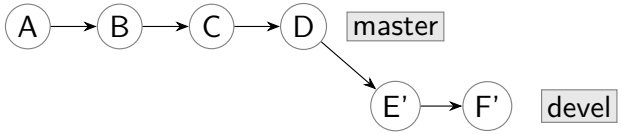
Rebaser des branches – rebase

rebaser X sur Y = déplacer à la suite de Y tous les commits de X
(changements depuis leur ancêtre commun)

- situation d'origine



- `git checkout devel`
- `git rebase master`
- situation après rebase



Dépôts distants et travail collaboratif

Dépôts distants – remote

Quand on **clone** un dépôt dans un répertoire,
le lien avec le dépôt distant (remote repository) est conservé :
`git remote -v`

Par défaut, le dépôt distant s'appelle origin.

Exemples de commandes `git remote`

- `git remote -v`
  Liste les dépôts distants connus, avec leurs URL
- `git remote set-url origin <url>`
  Change l'URL du dépôt distant nommé *origin*
- `git remote add upstream <url>`
  Déclare un nouveau dépôt distant nommé *upstream*

TP — Dépôts distants 1

1. Cloner un dépôt Git local dans un nouveau répertoire nommé *clone*.
2. Vérifier que le clone connaît bien sa source.
3. Ajouter le dépôt distant initial sous le nom *github*.
4. Modifier et commiter dans le dépôt source, puis créer une nouvelle branche *ramille*. Quel impact pour le clone ?
5. Commenter ce qu'annonce `git fetch origin`.
6. Que décrit `git branch -vv` ?

Branches distantes et suivi

`git fetch [origin]` récupère les commits et les branches distantes. Aucune modification des données locales.

`git branch -a` montre les branches locales et distantes.

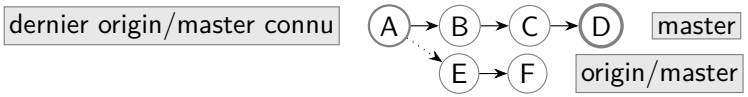
Suivi de branches

- Une branche locale peut suivre (track) une branche distante.
- Le suivi simplifie les échanges entre les deux branches.
- `git branch -vv` affiche les suivis.
- `git checkout branche-de-origin`
  Par défaut, si la branche locale n'existe pas, elle est créée et suit son homonyme du dépôt distant.
- `git branch --set-upstream-to=D/Y X`
  force la branche locale X à suivre Y du dépôt distant D.

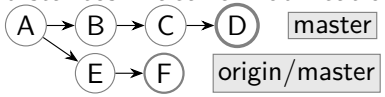
TP — Dépôts distants 2

1. Dans le répertoire *clone*, vérifier que la branche *ramille* est une référence distante, sans équivalent locale.
2. Créer la branche *ramille* qui suive celle du dépôt d'origine.
3. Dans le dépôt source, ajouter un commit à *ramille*. Le récupérer dans le dépôt clone.
4. Dans le clone, comparer les références *ramille* et *origin/ramille*.
5. Comment ajouter ce nouveau commit à la branche locale ?
6. Dans le second dépôt, ajouter un commit à *ramille*, puis utiliser `git push`. Quel impact sur le premier dépôt ?
7. Dans le second dépôt, créer et activer une nouvelle branche, puis utiliser `git push`. Pourquoi ces protestations de Git ?

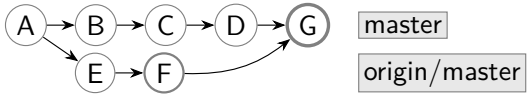
Résumé : réception



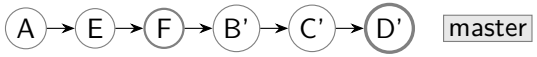
- `git fetch [origin]` récupère les commits et les branches distantes. Aucune modification des données locales.



- `git pull` récupère les commits de la branche distante suivie, puis les fusionne dans la branche locale (merge).



- `git pull --rebase` récupère les commits de la branche distante suivie, puis rebase la branche locale dessus.

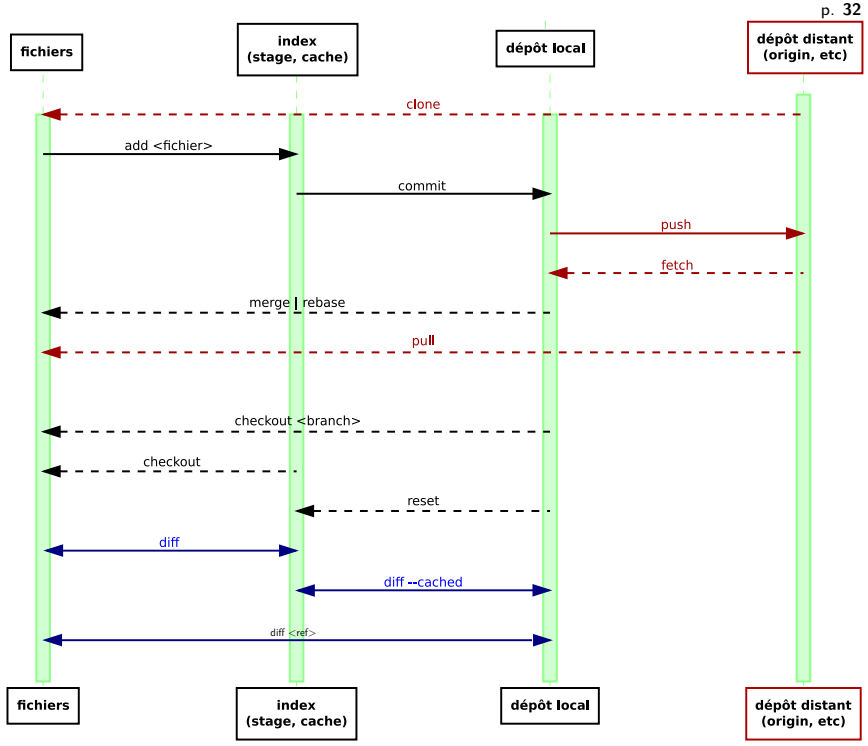


Résumé : émission

- `git push` envoie les commits de la branche locale sur la branche suivie.
  Si la branche distante n'est pas dans l'état attendu, il s'arrête.
- `git push myremote br-locale:br-distante`
  envoie les commits de *br-locale* sur la *br-distante* du dépôt *myremote*.
- `git push --force-with-lease`
  envoie les commits sur la branche suivie, même s'il ne sont pas dans le prolongement.

Attention, `git push --force` existe mais est très dangereux. `git push --force-with-lease` est suffisant en cas de *rebase* local, et il n'efface pas les commits déposés entre temps.

Compléments et recettes de Git



Stash

Mettre de côté les modifications locales.

1. `git stash`
   enregistre toutes les modifications des fichiers suivis, et rétablit ces fichiers à l'état de HEAD.
2. `git stash list`
3. `git stash pop` restaure l'état mis de côté et efface la sauvegarde.

Chercher

Chercher dans les métadonnées

1. `git log --grep=`*`motif`* `master..topic`
   cherche dans les messages de commits.
2. `git log --name-status --since="3 days"` Lister les commits récents avec leurs fichiers modifiés.
3. `git log --author=`*`motif`* `-- ce/répertoire/`

Chercher dans les fichiers

1. `git log -S`*`motif`*
   cherche les fichiers où une ligne contenant *motif* a été changée, dans les ancêtres du commit courant.
2. `git log -S`*`motif`* `--all`
   idem, mais cherche dans tous les commits.
   À compléter avec `git branch -a --contains` *`rév`* pour savoir à quelles branches appartient le commit trouvé.

Localiser la panne…et le suspect !

Recherche dichotomique

Quand ça marche plus alors que ça marchait…

1. `git bisect start`
2. `git bisect bad` marque le commit courant "buggé".
3. `git bisect good` *`dernière-rév-correcte`*
4. Itérer `git bisect bad|good` à chaque pas

Il est possible d'automatiser avec `git bisect run`.

Qui a écrit ça ?

1. `git blame --` *`fichier`*
   affiche le commit et l'auteur de la dernière modification, pour chaque ligne de fichier *fichier*.
2. `git blame -n -L 50,70 --` *`fichier`*
   Idem, mais restreint aux lignes 50 à 70, affichées numérotées.

Annuler, oublier, abandonner…

Annuler un ajout à l'index

`git reset -- <fichier>` est l'antidote de `git add`

Annuler un ancien commit

`git revert <rév>`
Nouveau commit qui annulera l'ancien

Oublier les modifications locales d'un fichier

`git checkout -- <fichier>`
Danger ! perte de fichiers non sauvegardés

Abandonner ses commits locaux

Par exemple sur *master*, en étant à jour avec `git fetch`

1. `git reset --hard origin/master`

Danger ! Perte de fichiers et de commits non sauvegardés

Réécrire l'historique

Attention, si vous modifiez des commits **déjà publiés**,
soit vous ne pourrez pas les publier,
soit vous passerez en force et les collègues vous maudiront.

Retoucher le dernier commit

Préparer le complément avec `git add [-i]`, puis
`git commit --ammend`

Modifier les commits

Pour modifier les commits descendant de *<rév>* :
`git rebase -i <rév>`

- réordonner
- fusionner (fixup)
- et bien plus, cf messages de Git

C'est utile quand on développe sur une *topic* branche, pour nettoyer avant de publier.

Informations utiles

Pour garder le contact :
francois.gannaz@silecs.info

Les documents utilisés sont disponibles en ligne :
http://silecs.info/formations/git-2017/

- Transparents
- Documents de référence

Licence Creative Commons By SA

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

© 2017 François Gannaz <francois.gannaz@silecs.info>