Cheat sheet Git

Les commandes de base pour survivre...

Git est un logiciel de gestion de versions décentralisé qui utilise un système de connexion pair à pair. Le code informatique développé est stocké non seulement sur l'ordinateur de chaque contributeur du projet, mais il peut également l'être sur un serveur dédié. C'est un outil de bas niveau, qui se veut simple et performant, dont la principale tâche est de gérer l'évolution du contenu d'une arborescence.

Site Web	`https://git-scm.com`
Documentation	`https://git-scm.com/doc`
Manuel de référence	`https://git-scm.com/docs`

---

repository (dépôt) : manière de nommer le dossier contenant le projet suivi par Git. Il existe deux types de dépôt : le dépôt local et le dépôt distant.

remote (lien) : manière de nommer le lien entre le dépôt distant et le dépôt local. Un dépôt Git local peut avoir plusieurs liens vers des dépôts Git distants. Par défaut, le lien entre un dépôt local et distant est nommé *origin*.

commit (état) : correspond à une arborescence de fichiers (tree) enrichie de méta-données comme un message de description, le nom de l'auteur, etc. Il pointe également vers un ou plusieurs objets commit parents pour former un graphe d'historiques. Un commit possède un identifiant unique.

branch (branche) : les branches sont souvent utilisées pour avancer dans une partie du projet sans impacter une autre partie.

Initialiser

Initialiser un dépôt Git

`git init`
Initialise un nouveau dépôt Git local dans un répertoire existant

`git clone [url]`
Récupère en local un dépôt Git distant à partir d'une [url]

Un dépôt local est composé de trois "arbres" gérés par Git. le premier est l'**espace de travail** qui contient réellement les fichiers. le second est un **Index** qui joue un rôle d'espace de transit pour les fichiers et enfin **HEAD** qui pointe vers le dernier **commit** réalisé.

Configurer

Configurer les informations de l'utilisateur

`git config --global user.name "[nom]"`
Définir un [nom] associé aux **commits** dans le graphe d'historiques

`git config --global user.email "[email]"`
Définir un [email] associé aux **commits** dans le graphe d'historiques

Connecter

Créer les liens entre les dépôts locaux et distants

`git remote`
Liste les liens existant entre les dépôts local et distant(s)

`git remote add [remote] [url]`
Ajoute un lien nommé [remote] entre le dépôt local et [url]

`git remote remove [remote]`
Supprime le lien nommé [remote]

Ajouter & valider

Ajouter et valider des changements

`git status`
Liste les fichiers modifiés depuis le dernier **commit**

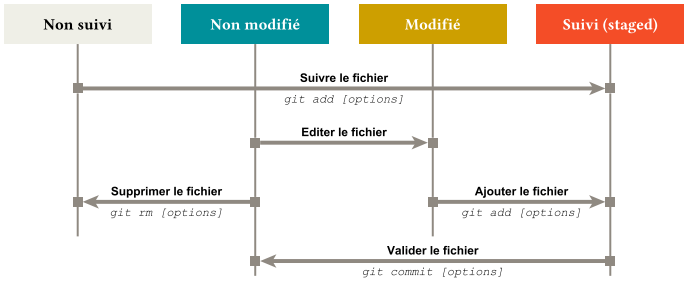
`git diff`
Affiche les changements par fichier depuis le dernier **commit**

`git add [fichier]`
Suit un [fichier] ; ajoute de(s) changement(s) sur le [fichier] (l'ajoute à l'**Index**)

`git add .`
Suit les fichiers ; ajoute des changements sur tous les fichiers modifiés (les ajoutent à l'**Index**)

`git commit -m "[message]"`
Valide, et annote avec [message], les changements qui sont ajoutés au **HEAD**

`git log`
Affiche tous les **commits** avec leurs identifiants



Mettre à jour & récupérer

Mettre à jour le dépôt local à partir d'un dépôt distant

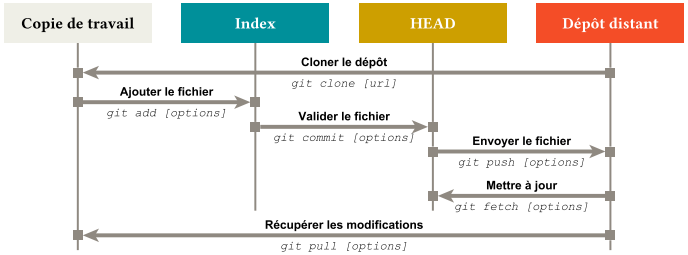
`git fetch [remote]`
Récupère le dernier historique du dépôt distant via le lien [remote]

`git pull [remote] [branch-name]`
Récupère et fusionne les **commits** du dépôt distant via le lien [remote] sur la branche [branch-name]

Envoyer

Envoyer des changements

`git push [remote] [branch-name]`
Envoie les **commits** via le lien [remote] sur la branche [branch-name]



Valider temporairement

Valider des changements temporairement

`git stash`
Valide temporairement les changements effectués

`git stash list`
Liste les validations temporaires effectuées

`git stash apply [stash-id]`
Applique la validation [stash-id] (si [stash-id] vide, applique la dernière)

`git stash pop [stash-id]`
Applique la validation [stash-id] et la supprime de la liste (si [stash-id] vide, applique la dernière)

`git stash drop [stash-id]`
Supprime la validation [stash-id] de la liste (si [stash-id] vide, applique la dernière)

Annuler

Annuler des changements effectués et non validés

`git checkout -- [file]`
`git restore --[options] [file]` (@Since Git 2.23)
Annule les changements locaux du [file] et revient à la version de [options = source/staged/worktree]

`git fetch [remote]`
`git reset --hard [remote]/[branch-name]`
Annule tous les changements locaux ; Récupère le dernier historique du dépôt distant via le lien [remote] sur la branche [branch-name]

Taguer

Taguer des validations

`git tag [tag-name] [commit-id]`
Tague le **commit** [commit-id] avec le nom [tag-name]

Ignorer

Ignorer des fichiers dans un projet Git

Créer un fichier **.gitignore** à la racine du projet. Chaque ligne de ce fichier contiendra un pattern, sous la forme d'une chaine de caractères, définissant les fichiers / dossiers à ignorer. **.gitignore** doit être **push** sur le dépôt distant.

```
#             # commentaire
*.temp        # pattern
fichier.out   # fichier
bin/          # dossier
```

Travail collaboratif

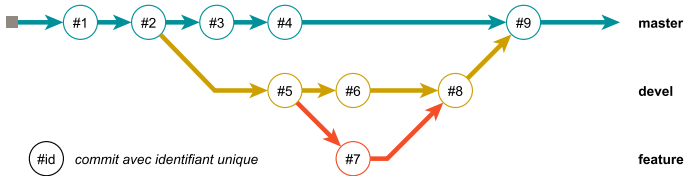
Travailler avec les branches

Le concept de branche est essentiel sous Git. Les branches permettent de :

- établir des règles de développement claires : définir les rôles de chaque branche, une pour la production, une pour les développements, etc. et garder le contrôle sur la structure du projet.
- structurer son workflow : créer une nouvelle branche pour chaque nouvelle fonctionnalité, pour facilement la supprimer ou revenir en arrière au cas-ou, tout en gardant un projet propre.
- expérimenter : créer une nouvelle branche pour expérimenter une nouvelle idée, la supprimer ou la mettre de côté si cela n'aboutit pas, sans perturber le développement du projet.

Lorsque un dépôt est initié, la branche par défaut est la branche **master**. Celle-ci ne peut être supprimée. Par convention, c'est cette branche qui accueille le code en *production*. Les développements se font sur une branche **devel**. Les ajouts dans cette branche seront ensuite fusionnés, au fur et à mesure, dans la branche principale.

Lors d'un projet collaboratif, utiliser les branches est primordial. En plus de structurer le projet, cela permet de travailler de manière concurrente en évitant le plus possible les conflits. Attention, une branche créée localement n'est pas disponible pour les autres tant qu'elle n'est pas envoyée vers le dépôt distant.



`git branch`
Liste toutes les branches (* indique la branche active)

`git branch [branch-name]`
Crée une nouvelle branche nommé [branch-name]

`git branch -d [branch-name]`
Supprime une branche locale nommé [branch-name]

`git push [remote] :[branch-name]`
Supprime une branche distante nommé [branch-name] via le lien [remote]

`git checkout [branch-name]`
`git switch [branch-name]` (@Since Git 2.23)
Se place dans la branche [branch-name]

`git push [remote] [branch-name]`
Envoie la branche [branch-name] via le lien [remote]

`git merge [branch-name]`
Fusionne la branche [branch-name] avec la branche actuelle

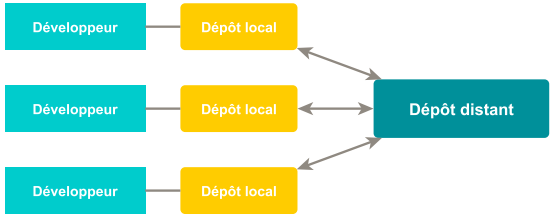
`git rebase [branch-name]`
Applique les **commits** de la branche actuelle sur la branche [branch-name]

`git log`
Affiche tous les **commits** de l'historique de la branche

Workflow

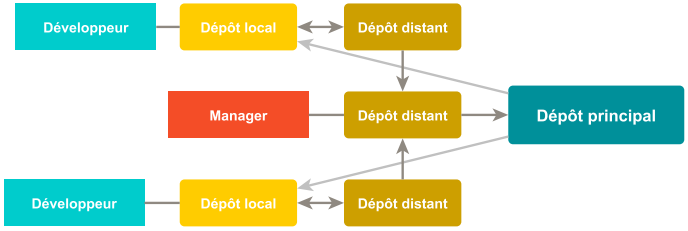
Centralized Workflow (type SVN)

Dans un workflow centralisé, tous les développeurs **clonent** et donc partagent le même dépôt distant. Ils peuvent donc envoyer des **commits** sur celui-ci. Cependant, Git n'autorisera pas à un développeur de **push** ces changements si quelqu'un d'autre à **push** des modifications entre temps (depuis le dernier **pull**/**fetch** de son dépôt local). Il est donc fortement conseillé d'utiliser le concept de **branch**. Ce type de workflow n'est, toutefois, pas recommandé pour du travail collaboratif.



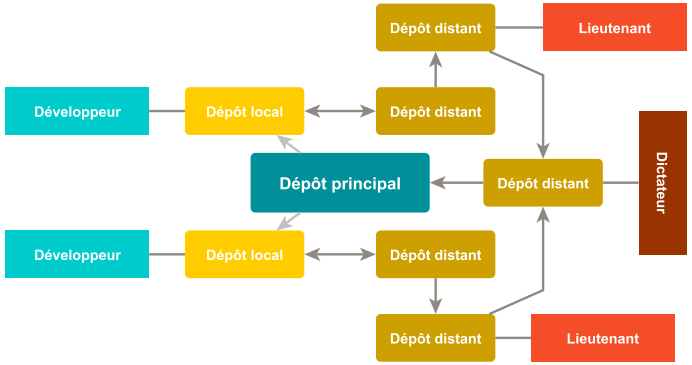
Integration Manager Workflow

Ce workflow intègre un *manager* d'intégration : une personne qui va contrôler les contributions, les valider (**commit**) et les envoyer (**push**) sur le dépôt principal distant. Pour se faire, les développeurs *forkent* le projet principal (consiste en une copie sur un dépôt distant), **commit** et **push** les changements dans leur dépôt local puis distant. Ils doivent ensuite réaliser une demande de mise à jour du dépôt principal (**pull request** / **merge request**) à partir de leurs dépôts publics. Chaque développeur doit ensuite **pull** les changements du dépôt principal et **rebase** leurs branches en conséquence. Ce type de workflow est le plus répandu dans le monde de l'open source et sur GitHub.



Dictator and Lieutenants Workflow

Pour des projets plus massifs, on ajoute deux nouveaux rôles à la place du manager : le *lieutenant* et le *dictateur*. Dans ce modèle, les lieutenants sont responsables d'une sous partie du projet. Les développeurs, assignés à ce sous projet, vont réaliser des **pull request** vers le dépôt du lieutenant qui rassemblera toutes les changements relatifs à ce sous système. Le lieutenant aura ensuite la charge de réaliser les **pull request** vers le dépôt du dictateur. Le dictateur **pull** tous les changements de tous les sous projets (venant uniquement des lieutenants) et a en charge de **push** vers le dépôt principal. Chaque développeur mettra à jour (**pull**) ses dépôts directement du dépôt principal.



Astuces

Trucs et astuces pour Git

`git config -global color.ui auto`
Active la coloration syntaxique pour Git (terminal)

`git commit -amend -m "[message]"`
Édite le [message] du dernier commit non envoyé

`git gc`
Nettoie et optimise le dépôt local

`git lg`
Affiche l'historique des **commits**

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

Emmanuel Hermellin
Ingénieur de recherche @ ONERA (DTIS/S2IM)
emmanuel.hermellin@onera.fr