Git - liste des commandes les plus fréquentes

```
git init                                  # initialiser un dépôt git
git clone <depot>                         # cloner un dépôt git déjà existant (depuis http ou ssh)

git config --global <opt>                 # color.diff auto,  color.status auto, color.branch auto, user.name <user>,
                                          # user.email <email> ; Ou éditer le fichier de configuration ~/.gitconfig

git status                                # afficher l'état courant du dépôt git
git diff [<file1> <file2> ...]            # afficher les changements non commités

git add <file1> <file2> ...               # ajouter les fichiers à l'index du dépôt git (ou les mettre à jour)
git rm  <file1> <file2> ...               # supprimer les fichiers de l'index du dépôt git

git commit -m <message>                   # commit les derniers modifications avec un message explicatif
git log [--oneline] [-n 25]               # afficher la liste de tous les commits exécutés sur le dépôt git

git checkout <fichier>                    # cela restaure le fichier tel qu'il était au dernier commit

git reset <commit>                        # revient au commit spécifié (HEAD, HEAD^, HEAD^^, id SHA..)
git reset --hard <commit>                 # cela annule sans confirmation tout votre travail

git push origin master                    # envoie les derniers commits au serveur
git pull origin master                    # récupère du serveur la dernière version du projet

git branch                                # afficher la liste des branches locales
git branch <nom_branche>                  # créer une nouvelle branche

git checkout <nom_branche>                # se positionner sur la branche spécifiée
git checkout -b <nom_branche>             # créer une nouvelle branche et se positionner dessus

git merge <nom_branche>                   # intégrer le travail fait sur la branche dans le projet principal
git branch -d <nom_branche>               # supprimer la branche après l'avoir intégrée au projet
git branch -D <nom_branche>               # supprimer la branche sans l'intégrer au projet (tout est perdu)

git blame <fichier>                       # voir qui a modifié quoi dans un fichier
git show <code-SHA>                       # voir les détails d'un commit en particulier
git show --name-only <code-SHA>           # voir la liste des fichiers modifiés dans ce commit

git stash                                 # mettre de coté les derniers modification non commités
git stash pop                             # ajoute ce qu'on avait mis de coté à la version courante du projet

git commit --amend                        # merge les derniers modifications dans le dernier commit
git commit --fixup <code-SHA>             # merge les dernières modifications dans un des commits précédents

git rebase -i [--autosquash] <code-SHA>   # modifier l'historique de vos commits
  e <code-SHA>                            # se positionner sur le commit pour l'éditer
  r <code-SHA>                            # se positionner sur le commit pour changer son message
  s <code-SHA>                            # fusionner ce commit avec le précédent et modifier leur msg
  supprimer une ligne                     # supprimer les modifications effectuées dans ce commit
  changer d'ordre entre les lignes        # changer l'ordre des commits dans l'historique
```

Git - logiciel de gestion de versions

Un **logiciel de gestion de versions** est devenu indispensable lorsqu'on travaille à plusieurs sur un même projet et donc sur le même code source

Ces outils suivent l'évolution de vos fichiers source et gardent les anciennes versions de chacun d'eux. Ils proposent en plus de nombreuses fonctionnalités qui vont vraiment vous être utiles tout au long de l'évolution de votre projet informatique :

- ils retiennent **qui a effectué chaque modification** de chaque fichier et pourquoi. Ils sont par conséquent capables de dire qui a écrit chaque ligne de code de chaque fichier et dans quel but ;
- si deux personnes travaillent simultanément sur un même fichier, ils sont capables **d'assembler** (de fusionner) **leurs modifications** et d'éviter que le travail d'une de ces personnes ne soit écrasé.

Il existe deux types principaux de logiciels de gestion de versions.

- logiciels **centralisés** : un serveur conserve les anciennes versions des fichiers et les développeurs s'y connectent pour prendre connaissance des fichiers qui ont été modifiés par d'autres personnes et pour y envoyer leurs modifications.
- logiciels **distribués** : il n'y a pas de serveur, chacun possède l'historique de l'évolution de chacun des fichiers. Les développeurs se transmettent directement entre eux les modifications, à la façon du peer-to-peer.

Le plus souvent on utilise un serveur qui sert de point de rencontre entre les développeurs. Le serveur connaît l'historique des modifications et permet l'échange d'informations entre les développeurs, qui eux possèdent également l'historique des modifications.

Il existe de nombreux logiciels de gestion de versions, comme SVN (Subversion), Mercurial et Git.

Git (prononcez « guite ») est un des plus puissants logiciels de ce genre. Il a l'avantage d'être à la fois flexible et pratique. Pas besoin de faire de sauvegarde du serveur étant donné que tout le monde possède l'historique des fichiers, et le serveur simplifie la transmission des modifications.

On retiendra surtout que Git :

- est très **rapide** ;
- sait travailler par **branches** (versions parallèles d'un même projet) de façon très flexible ;
- est assez **complexe**, il faut un certain temps d'adaptation pour bien le comprendre et le manipuler, mais c'est également valable pour les autres outils ;
- est à l'origine **prévu pour Linux**. Il existe des versions pour Windows mais pas vraiment d'interface graphique simplifiée. Il est donc à réserver aux développeurs ayant un minimum d'expérience et… travaillant de préférence sous Linux.

Une des particularités de Git, c'est l'existence de sites web collaboratifs basés sur Git comme **GitHub** et **Gitorious**. GitHub, par exemple, est très connu et utilisé par de nombreux projets : jQuery, Symfony, Ruby on Rails, … .C'est une sorte de **réseau social pour développeurs** : vous pouvez regarder tous les projets évoluer et décider de participer à l'un d'entre eux si cela vous intéresse. Vous pouvez aussi y créer votre propre projet : c'est gratuit pour les projets open source et il existe une version payante pour ceux qui l'utilisent pour des projets propriétaires. GitHub fournit le serveur où les développeurs qui utilisent Git se rencontrent. C'est un excellent moyen de participer à des projets open source et de publier votre projet !

Installer git sous Linux

Code : Console

```
sudo apt-get install  git-core  gitk
```

Cela installe 2 paquets :

- git-core : c'est git, tout simplement. C'est le seul paquet vraiment indispensable ;
- gitk : une **interface graphique** qui aide à mieux visualiser les logs. Facultatif.

Configurer git

Dans la console, commencez par envoyer ces trois lignes :

```
git config --global color.diff auto
git config --global color.status auto
git config --global color.branch auto
```

Elles activeront la couleur dans Git; à faire une fois et ça aide à la lisibilité des messages dans la console.

De même, il faut configurer votre nom (ou pseudo) :

```
git config --global user.name "votre_pseudo"
```

Puis votre e-mail :

```
git config --global user.email moi@email.com
```

Vous pouvez aussi éditer votre fichier de configuration .gitconfig situé dans votre répertoire personnel pour y ajouter une section alias à la fin :

```
vim ~/.gitconfig
[color]
	diff = auto
	status = auto
	branch = auto
[user]
	name = votre_pseudo
	email = moi@email.com
[alias]
	ci = commit
	co = checkout
	st = status
	br = branch
```

Ces alias permettent de raccourcir certaines commandes de Git. Ainsi, au lieu d'écrire `git checkout` , vous pourrez écrire si vous le désirez `git co` , ce qui est plus court.

Créer un nouveau dépôt ou cloner un dépôt existant

Pour commencer à travailler avec Git, il y a deux solutions :

- soit vous créez un **nouveau dépôt vide**, si vous souhaitez commencer un nouveau projet ;
- soit vous clonez un dépôt existant, c'est-à-dire que vous récupérez tout l'historique des changements d'un projet pour pouvoir travailler dessus.

Dans un logiciel de gestion de versions comme Git, **un dépôt représente une copie du projet.** Chaque ordinateur d'un développeur qui travaille sur le projet possède donc une copie du dépôt. Dans chaque dépôt, on trouve les fichiers du projet ainsi que leur historique.

Créer un nouveau dépôt vide

Commencez par créer un dossier du nom de votre projet sur votre disque.

```
cd /home/user/
mkdir mondossier
cd mondossier
```

Ensuite, initialisez un dépôt Git tout neuf dans ce dossier avec la commande :

```
git init            # créer un nouveau dépôt git dans le dossier où l'on se trouve
```

Cloner un dépôt existant

Cloner un dépôt existant consiste à **récupérer tout l'historique et tous les codes source d'un projet** avec Git. Pour trouver un dépôt Git, rendez-vous sur GitHub. Vous y trouverez de nombreux projets (certains n'utilisent pas GitHub directement mais on y trouve quand même une copie à jour du projet).

Prenons par exemple Symfony, la nouvelle version du framework PHP qui permet de créer des sites robustes facilement (il s'agit ici de la version *Symfony 2* en cours de développement).

Vous y voyez la liste des fichiers et des derniers changements ainsi qu'un champ contenant l'adresse du dépôt. En l'occurrence, l'adresse du dépôt de Symfony est :

```
http://github.com/symfony/symfony.git
```

Comme vous le voyez, on se connecte au dépôt en HTTP, mais il existe d'autres méthodes :
les **protocoles** **« git:// »** et **« ssh:// ».**

Le plus souvent on utilise SSH car il permet de chiffrer les données pendant l'envoi et gère l'authentification des utilisateurs.

Pour cloner le dépôt de Symfony, il suffit de lancer la commande suivante :

```
git clone http://github.com/symfony/symfony.git
```

Cela va créer un dossier « symfony » et y télécharger tous les fichiers source du projet ainsi que l'historique de chacune de leurs modifications !

Git compresse automatiquement les données pour le transfert et le stockage afin de ne pas prendre trop de place. Néanmoins, le clonage d'un dépôt comme ceci peut prendre beaucoup de temps.

Les messages suivants devraient apparaître dans la console :

```
Initialized empty Git repository in /home/mateo21/symfony/.git/
remote: Counting objects: 7820, done.
remote: Compressing objects: 100% (2490/2490), done.
remote: Total 7820 (delta 4610), reused 7711 (delta 4528)
Receiving objects: 100% (7820/7820), 1.40 MiB | 479 KiB/s, done.
Resolving deltas: 100% (4610/4610), done.
Checking out files: 100% (565/565), done.
```

Inutile d'essayer d'en comprendre le détail. Vous noterez toutefois que les **objets sont compressés avant l'envoi**, ce qui accélère le téléchargement. Une fois les fichiers reçus, Git les organise sur votre disque et vous voilà désormais en possession de tous les changements des fichiers du projet ainsi que de leur dernière version !

Vous pouvez ouvrir le dossier « symfony » et regarder son contenu, il y a tout le code source du projet. Le seul dossier un peu particulier créé par Git est un **dossier .git** (c'est un dossier caché situé à la racine du projet). Il contient l'historique des modifications des fichiers et la configuration de Git pour ce projet.

Sur **GitHub**, la plupart des dépôts sont en mode lecture seule (**read only**). Vous pouvez télécharger les fichiers, effectuer des modifications sur votre ordinateur (en local) mais pas les envoyer sur le serveur. Ceci permet d'éviter que n'importe qui fasse n'importe quoi sur les gros projets publics. Si vous faites une modification utile sur le projet (vous résolvez un bug par exemple) il faudra demander à l'un des principaux développeurs du projet de faire un **« pull »** , c'est-à-dire de télécharger vos modifications lui-même afin qu'il puisse vérifier si elles sont correctes.

Créer un nouveau dépôt vide qui servira de serveur

Si vous souhaitez mettre en place un **serveur de rencontre** pour votre projet, il suffit d'y faire un git clone ou un git init avec l'**option --bare**.

Cela aura pour effet de créer un dépôt qui contiendra uniquement le dossier .git représentant l'historique des changements (suffisant, car personne ne modifie les fichiers source directement sur le serveur).

```
git init --bare                    # => à exécuter sur le serveur.
```

Pour se connecter au serveur, la meilleure méthode consiste à utiliser SSH.

Ainsi, si vous voulez cloner le dépôt du serveur sur votre ordinateur :

```
git clone ssh://utilisateur@monserveur.domaine.com/chemin/vers/le/depot/git
```

Il faudra bien entendu vous identifier en entrant votre mot de passe (sauf si vous avez autorisé votre clé publique).

Modifier le code et effectuer des commits

Vérifier l'**état courant du dépôt git** avec **git status**

```
cd /home/mateo21/symfony
git status
# On branch master
nothing to commit (working directory clean)
```

Lorsqu'on travaille avec Git, on suit en général toujours les **étapes** suivantes :

- modifier le code source ;
- tester votre programme pour vérifier si cela fonctionne ;
- faire un commit pour « enregistrer » les changements et les faire connaître à Git ;
- recommencer à partir de l'étape 1 pour une autre modification.

Un commit représente donc un ensemble de changements.
Les commits sont là pour « valider » l'avancement de votre projet : n'en faites pas un pour chaque ligne de code modifiée, mais n'attendez pas d'avoir fait 50 modifications différentes non plus ! À vous de déterminer, dès que vos changements sont stables, quand vous devez faire un commit.

Après avoir modifié un fichier :

```
git status
# On branch master
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified: src/Symfony/Components/Yaml/Yaml.php
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git vous liste tous les fichiers qui ont changé sur le disque.
Il peut aussi bien détecter les modifications que les ajouts, les suppressions et les renommages.

Vous pouvez voir concrètement **ce que vous avez changé** en tapant **git diff** :

```
git diff
diff --git a/src/Symfony/Components/Yaml/Yaml.php b/src/Symfony/Components/Yaml/index fa0b806..77f9902
--- a/src/Symfony/Components/Yaml/Yaml.php
+++ b/src/Symfony/Components/Yaml/Yaml.php
@@ -19,7 +19,7 @@ namespace Symfony\Components\Yaml;
*/
class Yaml
{
- static protected $spec = '1.2';
+ static protected $spec = '1.3';
/**
* Sets the YAML specification version to use.
@@ -33,6 +33,8 @@ class Yaml
if (!in_array($version, array('1.1', '1.2'))) {
throw new \InvalidArgumentException(sprintf('Version %s of the YAML
}
++
$mtsource = $version;
self::$spec = $version;
}
```

Les lignes ajoutées sont précédées d'un **+** tandis que les lignes supprimées sont précédées d'un **-** . Normalement les lignes sont colorées et donc faciles à repérer.

Par défaut, Git affiche les modifications de tous les fichiers qui ont changé.

Vous pouvez demander à Git d'afficher seulement les changements d'un fichier précis, comme ceci :

```
git diff src/Symfony/Components/Yaml/Yaml.php
```

Effectuer un commit des changements

En faisant **git status** , vous devriez voir les fichiers que vous avez modifiés **en rouge**.
Cela signifie qu'ils ne seront pas pris en compte lorsque vous allez faire un commit.
Il faut explicitement préciser les fichiers que vous voulez « commiter ».

Pour cela, trois possibilités :

- faire git add pour ajouter les fichiers à la liste de ceux devant faire l'objet d'un commit, puis faire un git commit. Si vous faites un git status après un git add , vous les verrez alors **en vert** ;

  ```
  git add <nomfichier1> <nomfichier2>
  git commit
  ```

- faire git commit -a pour « commiter » tous les fichiers qui étaient listés dans git status dans les colonnes « Changes to be committed » et « Changed but not updated » ;

  ```
  git commit -a
  ```

- faire git commit nomfichier1 nomfichier2 pour indiquer lors du commit quels fichiers précis doivent être « commités ».

  ```
  git commit <nomfichier1> <nomfichier2>
  ```

J'utilise personnellement la **seconde solution** lorsque je veux « commiter » tous les fichiers que j'ai modifiés, et la **troisième solution** lorsque je veux « commiter » seulement certains des fichiers modifiés.

Faire appel à **git add** est indispensable lorsque vous venez de créer de nouveaux fichiers que Git ne connaît pas encore. Cela lui permet de les **prendre en compte pour le prochain commit**.

Lorsque la commande **git commit** est lancée, l'éditeur par défaut s'ouvre. Vous devez sur la première ligne taper un message (même sur plusieurs lignes) qui décrit à quoi correspondent vos changements. Si vous ne mettez **pas de message** de commit, celui-ci sera **annulé**.

Exemple de messages de commit d'une ligne corrects :

- « Améliore la visibilité des post-it sur le forum. » ;
- « Simplifie l'interface de changement d'avatar. » ;
- « Résout #324 : bug qui empêchait de valider un tutoriel à plusieurs ».

Comme vous pouvez le constater, on a tendance à écrire les **messages de commit au présent**. D'autre part, vous remarquerez que la plupart des projets open source sont écrits en anglais, donc il est fréquent de voir des messages de commit **en anglais**.

Vous pouvez également passer le message du commit directement dans la commande :

```
git commit -m "Améliore la visibilité des post-it sur le forum"
```

Une fois le message de commit enregistré, Git va officiellement sauvegarder vos changements dans un commit. Il ajoute donc cela à la liste des changements qu'il connaît du projet.

Annuler des commits

Il est possible à tout moment de consulter l'**historique des commits** : ce sont les logs.

Vous pouvez ainsi retrouver tout ce qui a été changé depuis les débuts du projet. Lorsque vous avez effectué un commit, vous devriez donc le voir dans git log :

```
git log

commit 227653fd243498495e4414218e0d4282eef3876e
Author: Fabien Potencier <fabien.potencier@gmail.com>
Date: Thu Jun 3 08:47:46 2010 +0200
[TwigBundle] added the javascript token parsers in the helper extension

commit 6261cc26693fa1697bcbbd671f18f4902bef07bc
Author: Jeremy Mikola <jmikola@gmail.com>
Date: Wed Jun 2 17:32:08 2010 -0400
Fixed bad examples in doctrine:generate:entities help output.
```

Chaque commit est numéroté grâce à un long numéro hexadécimal (**SHA**) comme 12328abf231da8e6... Cela permet de les identifier.

Vous pouvez parcourir les logs avec les touches **PageUp**, **PageDown** et les flèches directionnelles et quitter en appuyant sur la touche **Q**. Git utilise en fait le programme « less » pour paginer les résultats.

Utilisez l'option **-p** pour avoir le détail des lignes qui ont été ajoutées et retirées dans chaque commit :

```
git log -p
```

Utilisez l'option **--stat** pour avoir un résumé plus court des commits

```
git log --stat
```

Modifier le dernier message de commit

Si vous avez fait une faute d'orthographe dans votre dernier message de commit ou que vous voulez tout simplement le modifier, vous pouvez le faire facilement grâce à la commande suivante :

```
git commit --amend
```

Cette commande est généralement utilisée juste après avoir effectué un commit. Il est en effet impossible de modifier le message d'un commit lorsque celui-ci a été transmis à d'autres personnes.

Annuler le dernier commit (soft)

```
git reset HEAD^
```

Cela annule le dernier commit et revient à l'avant-dernier.

Pour indiquer à quel commit on souhaite revenir, il existe plusieurs notations :

- **HEAD** : dernier commit ;
- **HEAD^** : avant-dernier commit ;
- **HEAD^^** : avant-avant-dernier commit ;
- **HEAD~2** : avant-avant-dernier commit (notation équivalente) ;
- **d6d98923868578a7f38dea7...** : indique un numéro de commit précis ;
- **d6d9892** : indique un numéro de commit précis (notation équivalente à la précédente, souvent écrire les premiers chiffres est suffisant tant qu'aucun autre commit ne commence par les mêmes chiffres).

Seul le commit est retiré de Git ; vos fichiers, eux, restent modifiés.

Vous pouvez alors à nouveau changer vos fichiers si besoin est et refaire un commit.

Annuler tous les changement du dernier commit (hard)

```
git reset --hard HEAD^
```

Cela annulera sans confirmation tout votre travail !
Faites donc attention et vérifiez que c'est bien ce que vous voulez faire !

Normalement, Git devrait vous dire quel est maintenant le dernier commit qu'il connaît (nouveau HEAD) :

```
git reset --hard HEAD^
HEAD is now at 6261cc2 Fixed bad examples in doctrine:generate:entities help output
```

Annuler les modifications d'un fichier avant un commit

Si vous avez modifié plusieurs fichiers mais que vous n'avez pas encore envoyé le commit et que vous voulez restaurer un fichier tel qu'il était au dernier commit, utilisez git checkout :

```
git checkout <nomfichier>
```

Le fichier redeviendra comme il était lors du dernier commit.

Annuler/supprimer un fichier avant un commit

Supposons que vous veniez d'ajouter un fichier à Git avec git add et que vous vous apprêtiez à le « commiter ». Cependant, vous vous rendez compte que ce fichier est une mauvaise idée et vous voudriez annuler votre git add .

Il est possible de retirer un fichier qui avait été ajouté pour être « commité » en procédant comme suit :

```
git reset HEAD -- <fichier_a_supprimer>
```

Télécharger les nouveautés et partager votre travail

La commande git pull **télécharge les nouveautés depuis le serveur** :

```
git pull                    // ou git pull origin master (origin=nom du remote; master = nom de la branche
```

Deux cas sont possibles :

- soit vous n'avez effectué aucune modification depuis le dernier pull, dans ce cas la mise à jour est simple (on parle de mise à jour **fast-forward**) ;
- soit vous avez fait des commits en même temps que d'autres personnes. Les changements qu'ils ont effectués sont alors **fusionnés** aux vôtres **automatiquement**.

Si deux personnes modifient en même temps deux endroits distincts d'un même fichier, les changements sont intelligemment fusionnés par Git.

Parfois, mais cela arrive normalement rarement, deux personnes **modifient la même zone de code** en même temps. Dans ce cas, Git dit qu'il y a **un conflit** car il ne peut décider quelle modification doit être conservée; il vous indique alors le nom des fichiers en conflit.

Ouvrez-les avec un éditeur et recherchez une ligne contenant « <<<<<<<<< ».
Ces symboles délimitent vos changements et ceux des autres personnes.
Supprimez ces symboles et gardez uniquement les changements nécessaires, puis faites un nouveau commit pour enregistrer tout cela.

Vous pouvez **envoyer vos commits** avec la commande git push :

```
git push
```

Le changement vers le serveur doit être de type fast-forward car le serveur ne peut régler les conflits à votre place s'il y en a. Personne ne doit avoir fait un push avant vous depuis votre dernier pull.

Le mieux est de **s'assurer que vous êtes à jour** en faisant un pull avant de faire un push. Si le push échoue, vous serez de toute façon invités à faire un pull.

Il est recommandé de faire **régulièrement des commits**, **mais pas des push**. Vous ne devriez faire un push qu'une fois de temps en temps (pas plus d'une fois par jour en général). Evitez de faire systématiquement un push après chaque commit. Pourquoi ? Parce que vous perdriez la facilité avec laquelle il est possible d'annuler ou modifier un commit.

Un push est irréversible. Une fois que vos commits sont publiés, il deviendra impossible de les supprimer ou de modifier le message de commit ! Réfléchissez donc à deux fois avant de faire un push. C'est pour cela que je recommande de ne faire un push qu'une fois par jour, afin de ne pas prendre l'habitude d'envoyer définitivement chacun de vos commits trop vite.

Annuler un commit publié

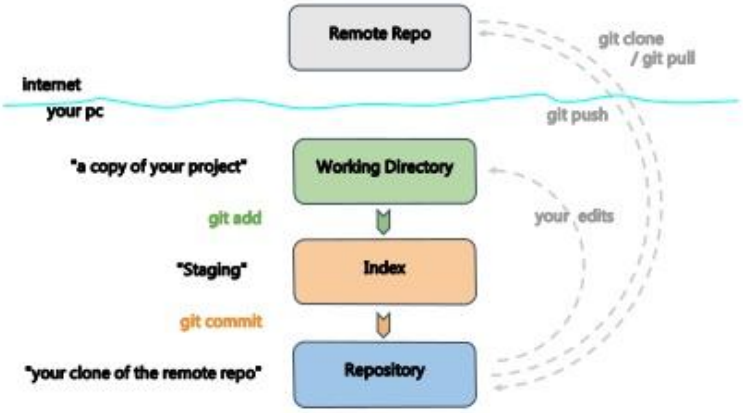
Dans le cas malheureux où vous auriez quand même envoyé un commit erroné sur le serveur, il reste possible de l'annuler… en créant un nouveau commit qui effectue l'inverse des modifications (souvenez-vous qu'**il n'est pas possible de supprimer un vieux commit envoyé**, on ne peut qu'en créer de nouveaux.). Les lignes que vous aviez ajoutées seront supprimées dans ce commit, et inversement.

Si vous vouliez annuler un commit 6261cc2..., il faut utiliser git revert qui va créer un commit inverse

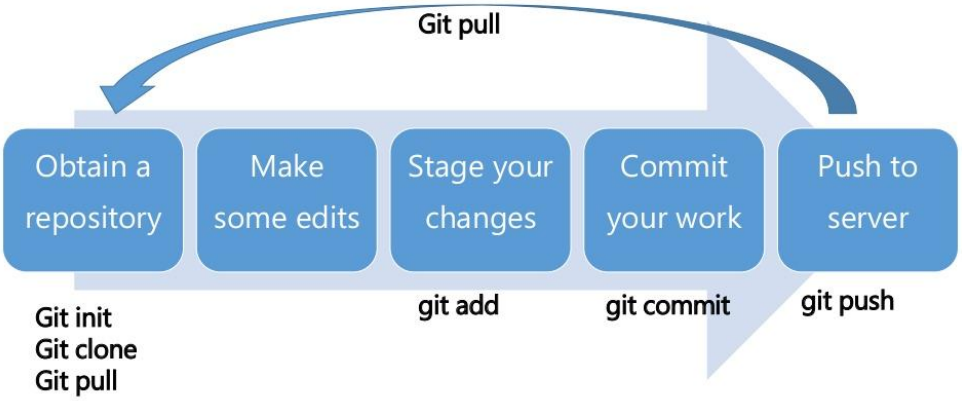
```
git revert 6261cc2
```

Il faut préciser l'ID du commit à « revert ». On vous invite à entrer un message de commit. Un message par défaut est indiqué dans l'éditeur. Une fois que c'est enregistré, le commit d'annulation est créé. Il ne vous reste plus qu'à vérifier que tout est bon et à le publier (avec un git push).

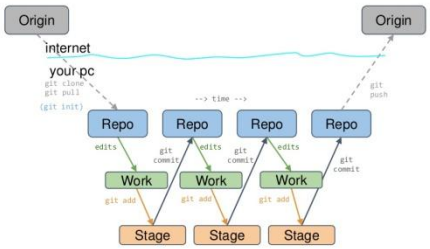
How it woks...

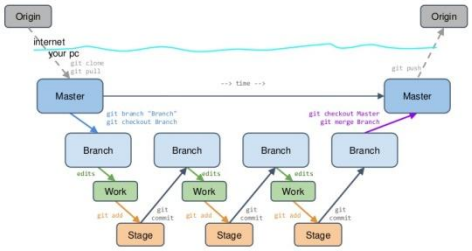


Git usages : Understanding Git Workflow



Branch Workflow



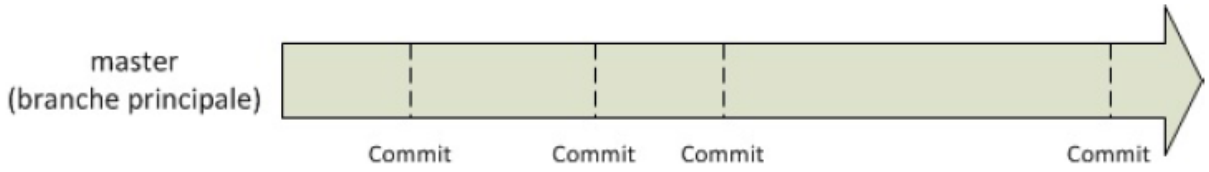


Travailler avec des branches

Les **branches** font partie du coeur même de Git et constituent un de ses principaux atouts. C'est un moyen de **travailler en parallèle** sur d'autres fonctionnalités. C'est comme si vous aviez quelque part une « copie » du code source du site qui vous permet de tester vos idées les plus folles et de vérifier si elles fonctionnent avant de les intégrer au véritable code source de votre projet.

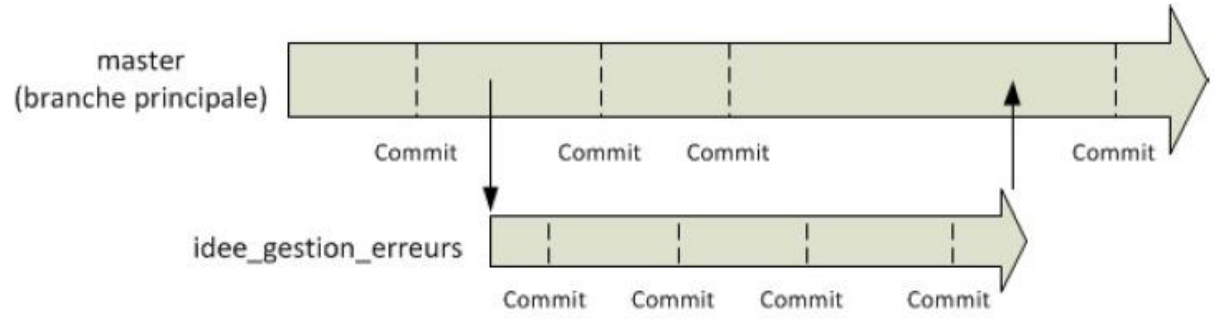
Dans Git, toutes les modifications que vous faites au fil du temps sont par défaut considérées comme appartenant à la **branche principale** appelée « **master** » :

On voit sur ce schéma les commits qui sont effectués au fil du temps.



Supposons que vous ayez une idée pour améliorer la gestion des erreurs dans votre programme mais que vous ne soyez pas sûrs qu'elle va fonctionner : vous voulez faire des tests, ça va vous prendre du temps, donc vous ne voulez pas que votre projet incorpore ces changements dans l'immédiat.

Il suffit de créer une branche, que vous nommerez par exemple « idee_gestion_erreurs », dans laquelle vous allez pouvoir travailler en parallèle :



Nous avons décidé de créer une nouvelle branche. Nous avons pu y faire des commits, mais cela ne nous a pas empêché de continuer à travailler sur la branche principale et d'y faire des commits aussi.

À la fin, mon idée s'est révélée concluante et j'ai donc intégré les changements dans la branche principale « master ». Mon projet dispose maintenant de mon idée que j'avais développée en parallèle. Tous les commits de ma branche se retrouvent fusionnés dans la branche principale.

Git gère tous ces problèmes pour vous. Au lieu de créer une copie des fichiers, il crée juste une branche « virtuelle » dans laquelle il retient vos changements en parallèle. Lorsque vous décidez de **fusionner une branche** (et donc de ramener vos changements dans « master » pour les valider), Git vérifie si vos modifications n'entrent pas en conflit avec des commits effectués en parallèle. S'il y a des conflits, il essaie de les résoudre tout seul ou vous avertit s'il a besoin de votre avis.

Vous pouvez même créer une **sous-branche** à partir d'une branche ! Si la sous-branche ne se révèle pas concluante, on peut la supprimer et aucun des commits intermédiaires ne sera finalement incorporé à la branche principale du projet.

Les branches locales

Tout le monde commence avec une seule branche « master » : c'est la branche principale. Jusqu'ici, vous avez donc travaillé dans la branche « master », sur le « vrai » code source de votre projet.

```
git branch                  // affiche la liste des branches locales
* master                    // l'étoile indique que c'est la branche sur laquelle vous êtes actuellement
```

Lorsque vous vous apprêtez à faire des modifications sur le code source, posez-vous les questions :

- « Ma modification sera-t-elle rapide ? » ;
- « Ma modification est-elle simple ? » ;
- « Ma modification nécessite-t-elle un seul commit ? » ;
- « Est-ce que je vois précisément comment faire ma modification d'un seul coup ? ».

Si la réponse à l'une de ces questions est « non », **vous devriez probablement créer une branche**.

Créer une branche est très simple, très rapide et très efficace. Il ne faut donc pas s'en priver.

Créez une branche pour toute modification que vous vous apprêtez à faire et qui risque d'être un peu longue. Au pire, si votre modification est plus courte que prévu, vous aurez créé une branche « pour pas grand-chose », mais c'est toujours mieux que de se rendre compte de l'inverse.

Supposons que vous vouliez « améliorer la page des options membres » du code de votre site. Vous n'êtes pas sûrs du temps que cela va prendre et vous risquez de faire plusieurs commits. Bref, il faut créer une branche pour cela.

```
git branch options_membres          // créer une branche avec le nom "options_membres"
```

Il est important de noter que cette branche est locale : vous seuls y avez accès.

Une fois la branche créée, vous devriez la voir quand vous tapez simplement git branch :

```
git branch
* master
  options_membres
```

Comme vous pouvez le voir, vous êtes toujours sur la branche « master ».

Pour aller sur la branche « options_membres », tapez ceci :

```
git checkout options_membres
```

! **git checkout** est utilisé pour changer de branche mais aussi pour restaurer un fichier tel qu'il était lors du dernier commit. La commande a donc un **double usage**.

Lorsque vous **changez de branche**, vous ne changez pas de dossier sur votre disque dur, mais Git change vos fichiers pour qu'ils reflètent l'état de la branche dans laquelle vous vous rendez. Les branches dans Git sont comme des dossiers virtuels **: vous « sautez » de l'un à l'autre avec la commande git checkout** . Vous restez dans le même dossier, mais Git modifie les fichiers qui ont changé entre la branche où vous étiez et celle où vous allez.

Faites maintenant des modifications sur les fichiers, puis un commit, puis d'autres modifications, puis un commit, etc. Si vous faites `git log` , vous verrez tous vos récents commits.

Maintenant, supposons qu'un bug important ait été détecté sur votre site et que vous deviez le régler immédiatement. Revenez sur la branche master avec `git checkout master`. Faites vos modifications, un commit, éventuellement un push s'il faut publier les changements de suite, etc. Ensuite, revenez à votre branche avec `git checkout options_membres` .

Sauvegarder les changements effectués dans une branche sans faire de commit

Si vous avez des changements non « commités » et que vous changez de branche, les fichiers modifiés resteront comme ils étaient dans la nouvelle branche.

Pour éviter d'avoir à faire un commit au milieu d'un travail en cours, tapez :

```
git stash
```

Vos fichiers modifiés seront sauvegardés et mis de côté. Maintenant, `git status` ne devrait plus afficher aucun fichier (on dit que votre working directory est propre).

On peut ensuite revenir sur notre branche et récupérer les changements qu'on avait mis de côté:

```
git stash pop                                    // ou apply ?
```

Fusionner les changements

Si vous faites un `git log` vous verrez que le commit que vous avez effectué sur la branche « master » n'apparaît pas. C'est en cela que les branches sont distinctes.

Lorsque vous avez fini de travailler sur une branche et que celle-ci est concluante, il faut « fusionner » cette branche vers « master » avec la commande `git merge`

Avec Git, on peut fusionner n'importe quelle branche dans une autre branche, bien que le plus courant soit de fusionner une branche de test dans « master » une fois que celle-ci est concluante.

Si vous avez fini votre travail dans la branche « options_membres » et que vous vouliez maintenant le publier, il faut fusionner le contenu de la branche dans la branche principale « master ».

```
git checkout master              // revenir sur la branche principale « master »
git merge options_membres        // intégrer le travail fait  dans « options_membres »
```

Pour récupérer les nouveaux commits depuis le serveur avec `git pull`, cela appelle deux commandes:

- `git fetch` : qui s'occupe du téléchargement des nouveaux commits,
- `git merge` : qui fusionne les commits téléchargés issus de la branche du serveur dans la branche de votre ordinateur !

Votre branche « options_membres » ne servant plus à rien, vous pouvez **la supprimer** :

```
git branch -d options_membres
```

Git vérifie que votre travail dans la branche « options_membres » a bien été fusionné dans « master ». Sinon, il vous en avertit et vous interdit de supprimer la branche (vous risqueriez sinon de perdre tout votre travail dans cette branche !).

Si vraiment vous voulez supprimer une branche même si elle contient des changements que vous n'avez pas fusionnés (parce que votre idée à la base était une erreur, ...), utilisez l'option -D (lettre capitale) :

```
git branch -D options_membres          /!\ Supprime la branche et perd tous les changements
```

Les branches partagées

Il est possible de travailler à plusieurs sur une même branche.

Pour lister toutes les branches que le serveur connaît :

```
git branch -r
origin/HEAD
origin/master
```

Il y a une seule branche sur le serveur (master).
Le serveur ne connaît que l'historique de la branche principale.

Si le serveur possède une autre branche et que vous souhaitez travailler dessus, il faut créer une copie de cette branche sur votre ordinateur, qui va « suivre » les changements sur le serveur.

```
git branch --track branchelocale    origin/brancheserveur  // paramètres : <local> <serveur>
git branch --track options_membres  origin/options_membres
```

Lorsque vous ferez un pull depuis la branche « options_membres », les changements seront fusionnés dans votre branche « options_membres » en local.

Il est donc important de savoir dans quelle branche vous vous trouvez avant de faire un pull. Un pull depuis la branche « master » met à jour votre branche « master » locale en fonction de ce qui a changé sur le serveur, et il en va de même pour n'importe quelle autre branche.

Il est possible d'ajouter des branches sur le serveur pour y travailler à plusieurs :

```
git push origin origin:refs/heads/nom_nouvelle_branche
```

Vous pouvez ensuite créer une branche locale qui « suit » la branche du serveur avec git branch --track.

Pour supprimer une branche sur le serveur :

```
git push origin :heads/nom_branche_a_supprimer
```

À noter que les « remote tracking branches » ne seront pas automatiquement supprimées chez les autres clients. Il faut qu'ils les suppriment manuellement à l'aide de la commande suivante :

```
git branch -r -d origin/nom_branche_a_supprimer
```

Résoudre les conflits

Exemple d'un fichier "hello.md" sur lequel on fait des modification à partir de la

- branche **master**
  Le fichier hello.md contient la phrase "Hello les amis!"
- d'une branche secondaire appelé "**modif**"
  Le fichier hello.md contient la phrase "Hello tout le monde!"

Au moment du merge de la branche "modif" sur la branche master il y aura un conflit.

```
git branch                          // s'assurer qu'on se situe dans la branche master
* master
  modif

git merge modif                     // merge le contenu de la branche modif sur la branche master
Auto-merging hello.md
CONFLICT (content): Merge conflict in hello.md
Automatic merge failed; fix conflicts and the commit the result.
```

En conséquence, le merge a échoué et le fichier hello.md a été modifié pour faciliter la correction du code

```
cat hello.md                        // afficher le contenu courant du fichier (après le merge échoué)

<<<<<<<<<< HEAD
Hello les amis!                     // le contenu du fichier dans la branche master (ici HEAD)
==========                                  // séparation de deux sources
Hello tout le monde!                // le contenu du fichier dans la branche modif
>>>>>>>>>> modif
```

Il faut supprimer les commentaires introduites automatiquement par git.
Il faut également supprimer le code qu'on veut pas conserver.

Par exemple, on va choisir de conserver uniquement le texte "Hello les amis!" et supprimer tout le reste.

```
cat hello.md
Hello les amis!
```

On vérifie l'état du dépôt git :

```
git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   hello.md

no changes added to commit (use "git add" and/or "git commit -a")
```

On ajoute le fichier modifié et on commit les derniers actions :

```
git add hello.md
git commit              // sans aucun paramètre (en l'occurrence sans message)
                        // un éditeur de texte va s'ouvrir avec le message généré automatiquement par git;
                        // on le laisse tel quel; on sauvegarde et on quitte l'éditeur de texte

// affiche : [master df17aa5] Merge branch 'modif'
```

On peut bien sur définir notre propre message de commit (en cas d'un merge plus compliqué).

Savoir qui a fait une modification

La commande `git blame` permet d'identifier la personne qui a modifié une ligne en particulier.

Exemple :

```
git blame liste.md

^05b1233 (Marc Gauthier 2014-08-08 00:31:02 +0200 1) # Une liste
^05b1233 (Marc Gauthier 2014-08-08 00:31:02 +0200 2)
30995548 (Marc Gauthier 2014-08-08 00:31:40 +0200 3) ## Un sous titre
30995548 (Marc Gauthier 2014-08-08 00:31:40 +0200 4)
^05b1233 (Marc Gauthier 2014-08-08 00:31:02 +0200 5) - a
^05b1233 (Marc Gauthier 2014-08-08 00:31:02 +0200 6) - c
775bece2 (Marc Gauthier 2014-08-08 00:31:18 +0200 7) - b
^05b1233 (Marc Gauthier 2014-08-08 00:31:02 +0200 8) - d
30995548 (Marc Gauthier 2014-08-08 00:31:40 +0200 9) - e
```

Première colonne : les premiers caractères de l'identifiant du code SHA des commits

La commande `git show` permet de voir les détails d'un commit en particulier :

```
git show 775bece2

commit 775bece2019cece.....
author: Marc Gauthier <marcg.gauthier@gmail.com>
Date:   Friday Aug 8 00:31:18 2014 +0200

      inversé b et c

diff --git a/liste.md b/liste.md
index 2159c5f..a6219df 100644
--- a/liste.md
+++ b/liste.md
@@ -1,6 +1,6 @@
  # Une liste

 - a
-- b
 - c
+- b
 - d
```

Ignorer des fichiers

Pour des raisons de sécurité et de clarté, il est important d'ignorer certains fichiers dans Git, tels que :

- les fichiers de configuration (config.xml, databases.yml, .env...)
- les fichiers et dossiers temporaires (tmp, temp/...)
- les fichiers inutiles comme ceux créés par votre IDE ou votre OS (.DS_Store, .project...)

Le plus crucial est de ne **JAMAIS versionner une variable de configuration**, que ce soit un mot de passe, une clé secrète ou quoi que ce soit de ce type. Versionner une telle variable conduirait à une large faille de sécurité, surtout si vous mettez votre code en ligne sur GitHub !

Si le code a déjà été envoyé sur GitHub, partez du principe que quelqu'un a pu voir vos données de configuration et mettez-les à jour (changez votre mot de passe ou générez une nouvelle clé secrète).

Par exemple, si on veut ignorer un fichier appelé "mon_fichier" lorsqu'on demande l'état du dépôt git, il faudra introduire son nom dans un fichier appelé ".gitignore".

```
vim .gitignore
mon_fichier                          # introduire le nom du fichier qu'on veut ignorer dans git
```

En conséquence, la commande `git status` ne proposera plus d'ajouter le fichier "mon_fichier" à l'index.

Contribuer à des projets open source

Sur GitHub :

- aller sur la page GitHub du projet auquel je veux participer
  (vérifier s'il existe une section "Contributing" dans le readme du projet)
- créer une copie de ce projet dans mon compte personnel en cliquant sur le lien **Fork**
- cloner le projet depuis mon répertoire GitHub sur ma machine
- créer une nouvelle branche : `git checkout -b my-new-feature`
- commit les changements dans la branche : `git commit -am "Add some features"`
- push sur la branche : `git push origin my-new-feature`
- demander un pull request (sur le site de GitHub)

Autres fonctionalités de GIT

- **Tagger une version**

  Donner un alias précis pour référencier un commit sous ce nom.

  Par exemple : pour dire « À tel commit correspond la version 1.3 de mon projet ».

  Cela permettra à d'autres personnes de repérer la version 1.3 plus facilement.

```
git tag <NOMTAG> <IDCOMMIT>
git tag v1.3 2f7c8b3428aca535fdc4...
git push --tags                                  // pour envoyer les tags lors d'un push
git tag -d v1.3                                  // pour supprimer un tag crée
```

- **Rechercher dans les fichiers source**

  Comme Git connaît tous les fichiers source de votre projet, il est facile de faire une recherche à l'intérieur de tout votre projet.

  Par exemple, si vous voulez connaître les noms des fichiers qui contiennent le mot TODO :

```
git grep "TODO"
```

  Si vous voulez connaître les numéros des lignes qui contiennent le mot que vous recherchez :

```
git grep -n "TODO"
```

  Git accepte également les expressions régulières pour les recherches.