# [Utilisez Git et GitHub pour vos projets de développement](https://openclassrooms.com/fr/courses/5641721-utilisez-git-et-github-pour-vos-projets-de-developpement)

Git permet de suivre les modifications et organiser votre projet. C’est un outil essentiel, que vous travailliez **seul**, en **équipe**, ou même sur **un projet en open source**!

Dans ce cours, vous installerez et configurerez **Git** et son ami **GitHub**. Vous découvrirez les **commandes de base** avant d'apprendre à corriger vos erreurs simplement et efficacement. Vous découvrirez la **structure de l'arbre Git** et comment garder vos **branches** propres. Enfin, vous apprendrez à utiliser **GitFlow** et à gérer les demandes de **pull**.

Un contrôleur de versions est un programme qui permet aux développeurs de conserver un historique des modifications et des versions de tous les fichiers.

Dans ce cours, nous allons apprendre à utiliser le contrôleur de versions Git

## Comment l’utiliser dans des projets open source ?

*Open source* signifie que le code source d'un logiciel est public et accessible. Le logiciel en question peut alors être modifié et diffusé par n'importe quel individu.

**Git est l'outil qui nous permet de créer un dépôt local et de gérer les versions de nos fichiers, alors que GitHub est un service en ligne qui va héberger notre dépôt, qui sera du coup distant (puisqu'il ne sera pas sur notre machine).**

Un dépôt Git est un entrepôt virtuel de votre projet. Il vous permet d'enregistrer les versions de votre code et d'y accéder au besoin.

**L’onglet Pull requests**, quant à lui, permet de réaliser des demandes de pull. Les demandes de pull (extractions) vous permettent d'informer les autres sur les modifications que vous avez appliquées à une branche d'un référentiel sur GitHub. Une fois qu'une demande d'extraction est ouverte, vous pouvez discuter et examiner les modifications éventuelles avec les collaborateurs, et ajouter des validations de suivi avant que vos modifications ne soient fusionnées dans la branche de base.

Un des derniers points importants sur GitHub est la fonctionnalité **Explore**.

Via Explore, vous pourrez trouver de nouveaux projets open source intéressants sur lesquels travailler, en parcourant les projets recommandés, en vous connectant à la communauté GitHub et en recherchant des référentiels par sujet ou par libellé.

**Créez votre propre dépôt**

Pour mettre votre projet sur GitHub, vous devez créer un référentiel dans lequel il pourra être installé.

Cliquez sur le "+" dans le coin supérieur droit, pour faire apparaître l’option New repository.

### Initialisez Git

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

$ git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)

Si vous souhaitez par contre, pour **un projet spécifique**, changer votre nom d’utilisateur, vous devrez repasser cette ligne mais sans le --global.

Afin de vérifier que vos paramètres aient bien été pris en compte, et vérifier les autres paramètres, il suffit de passer la commande  git config --list

Par défaut, Git utilisera Vim comme éditeur et Vimdiff comme outil de merge. Vous pouvez les modifier en utilisant :

$ git config --global core.editor notepad++

$ git config --global merge.tool vimdiff

Nous allons **créer notre fameux dépôt local**. Pour ce faire, deux solutions possibles :

* créer un dépôt local vide pour accueillir un nouveau projet ;
* cloner un dépôt distant, c’est-à-dire rapatrier tout l’historique d’un dépôt distant en local, afin de pouvoir travailler par 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.

Accédez à votre dossier, et lancez la ligne ci-dessous dans Git Bash en ciblant ce dossier :

johndoe ~

$ cd Documents/Fichiers/Git/PremierProjet

johndoe ~/Documents/Fichiers/Git/PremierProjet

**$ git init**

Initialized empty Git repository in c:/users/JohnDoe/Documents/Fichiers/Git/PremierProjet/

Votre dossier n’a rien de nouveau, mais c’est normal. :) Vous avez pourtant bien initialisé votre dépôt Git. **Un dossier caché .git a été créé** ! Vous pouvez l'afficher en allant dans Affichage => Éléments masqués.

1) Créer le répertoire

2) S’y placer via le bash

3) ‘git init’

## Utilisez les commandes de base de Git !

**Comment accéder à un dépôt distant et le cloner en local :**

1) cliquer sur le dépôt

2) cliquer sur Clone or download

3) copier le lien

4) retourner dans git bash et taper : «  **git remote add nomDépot lien**» 🡺 Cette ligne ne permet pas de cloner le dépôt, mais permet de dire au dépôt que l’on pointe vers le dépôt distant.

5) Clonez le dépôt en local

Maintenant quenotre dépôt local pointe sur le dépôt distant, nous allons cloner son contenu et le dupliquer en local. Afin de réaliser le clonage, nous allons utiliser la commande :

**git clone https://github.com/OpenClassrooms-Student-Center/ProjetOpenSource.git**

🡺 Le dépôt distant est cloné en local

## Le Système de branche de Git

Les différentes branches correspondent à des copies de votre code principal à un instant T, où vous pourrez tester toutes vos idées les plus folles sans que cela impacte votre code principal.

Sous Git, la branche principale est appelée la **branche master**. C’est celle-ci, où au final, vous aurez à la fin toutes vos modifications. Le but est de ne surtout pas réaliser les modifications directement sur cette branche, mais de réaliser les modifications sur d’autres branches, et après tests, les intégrer sur la branche master.

**Pour connaître les branches présentes dans notre projet, il faut taper la ligne de commande :**

git branch

*Je vous conseille fortement de créer une branche si votre modification va être longue, qu’elle peut avoir des impacts, qu’elle n’est pas simple ou que vous ne voyez pas tout de suite comment faire la modification.*

Créer une branche

🡺 **git branch nomBranche**

A master branch is created after our first commit. Let’s commit a file and check the branch list:

$ touch test.txt

$ git add test.txt

$ git commit -m "First commit"

[master (root-commit) dffc881] First commit

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 test.txt

$ git branch

\* master

## Basculer sur une branche :

🡺 **Git checkout branche**

 Vous pouvez désormais réaliser votre évolution sans toucher à la branche master qui abrite votre code principal qui fonctionne. Vous pouvez rebasculer si besoin à tout moment sur la branche master, sans impacter les modifications de la branche Cagnotte.

## Réalisez un commit

Un commit est tout simplement un enregistrement de votre travail à un instant T sur la branche courante où vous êtes.

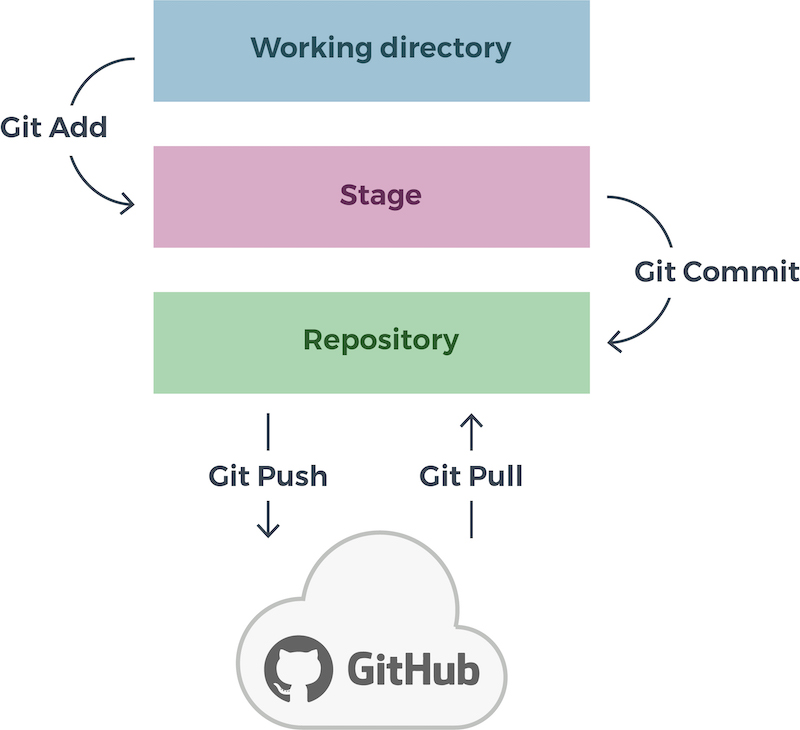
**git commit -m “message”**

## Faire un Push

La commande **Git push** permet d'envoyer les modifications que l'on a réalisées en local sur le dépôt à distance, alors que la commande **Git pull** permet de récupérer en local le projet distant ! Nous reparlerons de ces deux commandes, plus en détail, un peu plus tard dans le cours !

**Git gère les versions de vos travaux locaux à travers 3 zones locales majeures :**

* **le répertoire de travail (working directory/WD) ;**
* **l’index, ou *stage* (nous préférerons le second terme) ;**
* **le dépôt local (Git directory/repository).**



L'index, ou *stage*, désigne tous les fichiers modifiés que vous souhaitez voir apparaître dans votre **prochain commit**. C'est avec la fonction  **git add**  que l'on ajoute un fichier au stage.

Le dépôt local est **l'historique de l'ensemble de vos actions** (commits, configurations...). L'archivage se fait principalement avec la commande  **git commit**.  Il est possible d'accéder à cet historique en faisant un  **git reflog**  qui affichera toutes vos actions et leurs SHA. Le **SHA,** c'est ce grand code qui vous permettra de revenir à un commit exact. C'est l’identifiant de votre action !

### Supprimer une branche

**🡪 git branch -d nomBranche**

Attention, si toutefois vous avez déjà fait des modifications dans la branche que vous souhaitez supprimer, il faudra soit faire un commit de vos modifications, soit mettre vos modifications de côté, et ça, je vous l'expliquerai un peu plus tard ; soit forcer la suppression en faisant :

**git branch -D brancheTest**

## Créer le master

**Avant de créer une branche, vous devez créer votre branche principale (master)**. Pour créer la branche master, vous devez simplement ajouter un fichier et le commiter.

Créez un fichier "PremierFichier.txt" dans votre répertoire Test, et ajoutez-le avec la commande :

git add PremierFichier.txt

git commit -m « messag »

On vous demande alors d'indiquer le message du commit puis de valider. Pour valider le message, une fois que vous l'avez écrit, appuyez sur Echap (votre curseur va basculer sur la dernière ligne) et tapez  **:x**. Cette commande va sauvegarder et quitter l'éditeur des messages de commit.

## J’ai modifié la branche principale

Vous avez modifié votre branche master avant de créer votre branche et vous n'avez pas fait le commit. Ce cas est un peu plus simple. Nous allons faire ce qu'on appelle une *remise*. La remise va permettre de mettre vos modifications de côté, le temps de créer votre nouvelle branche et ensuite appliquer cette remise sur la nouvelle branche.

Vous pouvez à tout moment voir à quel état sont vos fichiers en faisant :

**git status**

Nous allons donc créer une remise. (on déplace ce qu’il y a à commit dedans afin de le déplacer ensuite dans une branche que l’on va créer)

**git stash**

Et finalement, nous allons pouvoir appliquer la remise, afin de récupérer nos modifications sur notre nouvelle branche.

**git stash apply**

Cette commande va appliquer la dernière remise qui a été faite. Si pour une raison ou une autre, vous avez créé plusieurs remises, et que la dernière n'est pas celle que vous souhaitiez appliquer, pas de panique, il est possible d'appliquer une autre remise.

Nous allons d'abord regarder la liste de nos remises. Pour ce faire :

**git stash list**

Il suffira alors d'appeler la commande git stash  en indiquant l'identifiant.

**git stash apply stash@{0}**

### Maintenant, admettons que vous ayez réalisé vos modifications et qu'en plus vous ayez fait le commit.

Nous allons devoir aller **analyser vos derniers commits** avec la fonction  **git log**, afin de pouvoir récupérer l'identifiant du commit que l'on appelle couramment le *hash*. Par défaut, git log  va vous lister par ordre chronologique inversé tous vos commits réalisés.

Maintenant que vous disposez de votre identifiant, gardez-le bien de côté. Vérifiez bien que vous êtes sur votre branche master et réalisez la commande suivante :

**git reset --hard HEAD^**

Cette ligne de commande va permettre de **supprimer de la branche master votre dernier commit**.  Le Head^ indique que c'est bien le dernier commit que nous voulons supprimer.

Nous allons maintenant créer notre nouvelle branche.

**git branch brancheCommit**

Nous allons basculer sur cette branche.

**git checkout brancheCommit**

Maintenant que nous sommes sur la bonne branche, nous allons de nouveau faire un  **git reset**, mais celui-**ci va permettre d'appliquer ce commit sur notre nouvelle branche** ! Il n'est pas nécessaire d'écrire l'identifiant en entier. Seuls les 8 premiers caractères sont nécessaires.

**git reset --hard ca83a6df**

git

## Je souhaite changer le message de mon commit

Attention ! Cette commande va fonctionner pour le dernier commit réalisé !

**git commit --amend -m "Votre nouveau message de commit"**

Et on vérifie avec  git log

## J’ai oublié un fichier dans mon dernier commit

Nous allons dans un premier temps ajouter notre fichier, et dans un deuxième temps réaliser le  git --amend.

git add FichierOublie.txt

**git commit --amend --no-edit**

Votre fichier a été ajouté à votre commit et grâce à la commande --no-edit que nous avons ajoutée, nous n'avons pas modifié le message du commit.

## Corrigez vos erreurs sur votre dépôt distant

#### Corrigez vos erreurs en local et à distance

**vous avez pushé des fichiers erronés**. Le problème, c'est que maintenant ce n'est plus que sur votre dépôt local, mais à disposition de tout le monde. La première chose à faire est de prévenir vos collaborateurs

Il est possible d'annuler son commit public avec la commande **Git revert**. L'opération Revert annule un commit en créant un nouveau commit. C'est une méthode sûre pour annuler des changements, car elle ne risque pas de réécrire l'historique du commit.

**git revert HEAD^**

Par conséquent, il vaut mieux utiliser  git revert  pour annuler des changements apportés à une branche publique, et git reset  pour faire de même, mais sur une branche privée.

*Gardez à l'esprit que Git revert sert à annuler des changements commités, tandis que Git reset HEAD permet d'annuler des changements non commités.*

### L'accès à distance ne fonctionne pas

Git base toute sa gestion d’authentification sur le mécanisme **des clés SSH.**

Nous allons maintenant générer notre duo de clés SSH :

Dans Git Bash, exécutez la commande :

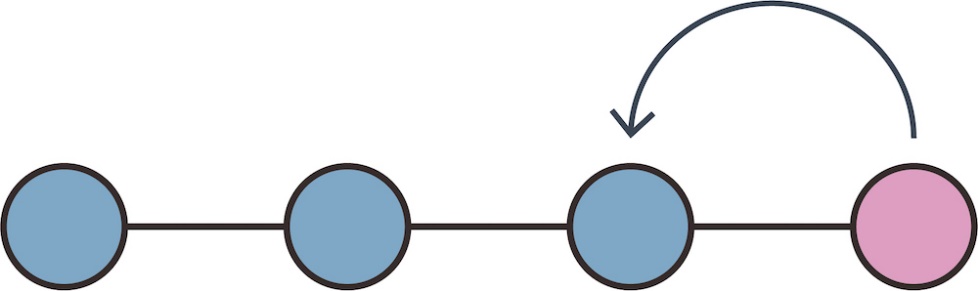
$ ssh-keygen -t rsa -b 4096 -C "johndoe@example.com"

## Modifiez vos informations d'identification et supprimez la clé

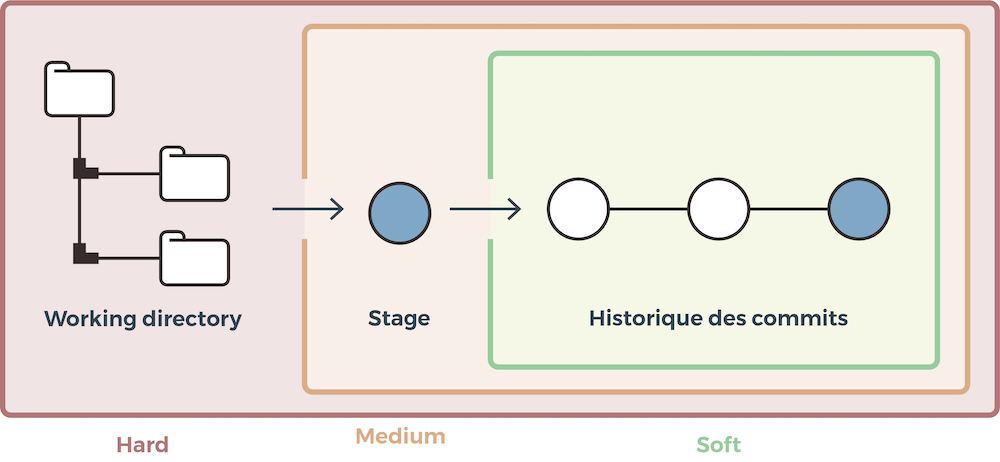
Ajoutons maintenant notre clé à notre compte GitHub.

Settings -> SSH & GPG 🡪 New SSH

### Utilisez Git reset



La commande  **git reset**  est un outil complexe et polyvalent pour **annuler les changements**. Elle peut être appelée de trois façons différentes, qui correspondent aux arguments de ligne de commande **--soft, --mixed et --hard**.



**git reset notreCommitCible --hard**

Cette commande va permettre de **revenir à n'importe quel commit** mais en oubliant absolument tout ce qu'il s'est passé après !

Le  **git reset --mixed**  va permettre de revenir juste après votre dernier commit ou le commit spécifié, sans supprimer vos modifications en cours. Il va par contre créer un HEAD détaché. Il permet aussi, dans le cas de fichiers indexés mais pas encore commités, de désindexer les fichiers.

Nous avons enfin le  git reset --Soft. Le **git reset --Soft** permet juste de se placer sur un commit spécifique afin de voir le code à un instant donné ou créer une branche partant d'un ancien commit. Il ne supprime aucun fichier, aucun commit, et ne crée pas de HEAD détaché.

### Conflit entre fichier lors du commit

Pour cela, rien de plus simple, vous allez devoir ouvrir un éditeur. Si vous utilisez VIM, vous pouvez l'ouvrir avec la ligne de commande :

**vim helloworld.php**

Sinon, vous ouvrez votre éditeur habituel. Vous allez devoir maintenant régler les conflits en comparant les deux fichiers HelloWorld.php et en choisissant quelles modifications vous gardez en fusionnant les fichiers.

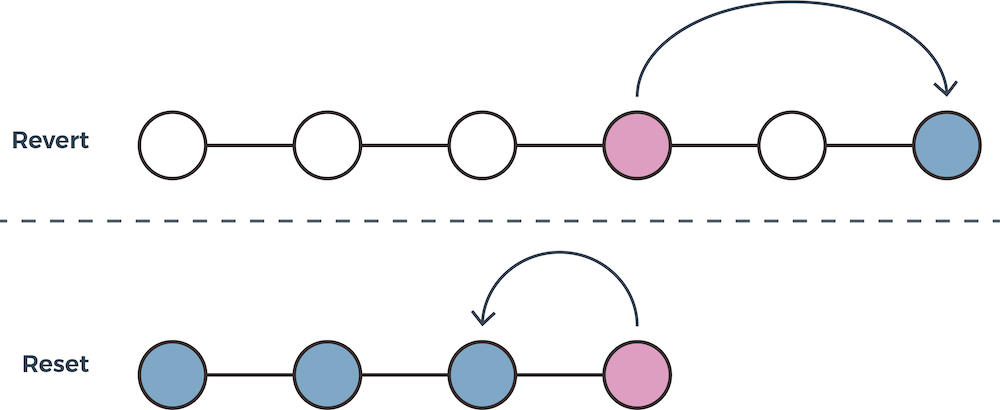
Maintenant que vous avez résolu le conflit, il vous reste à le dire à Git !

**git commit**

Git va détecter que vous avez résolu les conflits et va vous proposer un message de commit. Vous pouvez bien entendu le modifier.

### La machine à remonter le temps ! J’ai ajouté le mauvais fichier au commit

La différence entre Revert et Reset est que Reset va revenir à l'état précédent sans créer un nouveau commit, alors que Revert va créer un nouveau commit.



**git revert HEAD**

Une fois, votre commit "annulé", vous allez pouvoir enlever votre fichier, et réaliser de nouveau votre commit.

## Corrigez un commit raté

### Trou de mémoire ? Git reflog

L'objectif d'un système de contrôle de versions est d'enregistrer les changements apportés à votre code. Il vous permet de **consulter l'historique de votre projet** pour voir qui a contribué à quoi, de déterminer où des bugs ont été introduits et d'annuler les changements problématiques.

Par défaut,  **git log**   **énumère en ordre chronologique inversé** les commits réalisés. Cela signifie que les commits les plus récents apparaissent en premier. Cette commande affiche chaque commit avec son identifiant SHA, l'auteur du commit, la date et le message du commit.

**git reflog**  va loguer les commits, mais aussi toutes les autres actions que vous avez pu faire en local. Git reflog va enregistrer vos commits, vos modifications de messages, vos merges, vos resets, enfin tout, quoi. :-°  Ce qui est très pratique, c'est que comme Git log, Git reflog va afficher un identifiant SHA-1 pour chaque action. Il est donc très facile de revenir à une action donnée **grâce au SHA.** Cette commande, c'est votre joker, elle assure votre survie en cas d'erreur.  Pour revenir à une action donnée, on prend les 8 premiers caractères de son SHA et on fait :

git checkout e789e7c

## Le git blame

🡺 connaître à l'intérieur d'un fichier **qui a modifié chaque ligne**.

La commande  git blame  permet **d’examiner le contenu d’un fichier ligne par ligne** et de déterminer la date à laquelle chaque ligne a été modifiée, et le nom de l’auteur des modifications.

**git blame monFichier.php**

git blame  va afficher pour chaque ligne modifiée, son ID, l'auteur, l'horodatage, le numéro de la ligne et le contenu de la ligne. Avec git blame, vous devenez un super espion de votre projet !

## Le cherry pick

🡺 récupérer le/les commits qui nous intéressent

Parfois, vous ne voulez pas fusionner une branche entière dans une autre et vous n'avez besoin que de choisir un ou deux commits spécifiques. Ce processus s'appelle**cherry-pick** ! Attention, Git cherry-pick n'est pas très apprécié dans la communauté des développeurs ! En effet, cette commande va **dupliquer des commits existants**. Il sera préférable, si possible, de réaliser un merge.

# La structure du fichier GIT

Accéder à l’arbre de git 🡪 git log

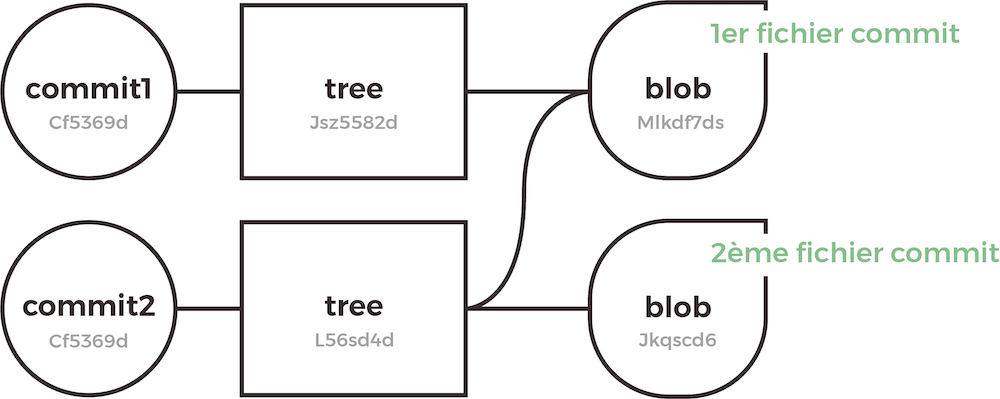
Il existe trois types d'objets principaux et un qui est un peu plus secondaire.

Les trois principaux types d'objets sont :

* le "**tree**" ou *l'arbre Git* qui est une forme de répertoire. Il va référencer une liste de trees et de blobs (sous-répertoires et fichiers) ;
* le "**commit**" qui va pointer vers un arbre spécifique et le marquer, afin de représenter son état à un instant donné ;
* Le "**blob**" qui représente en général un fichier (*Binary Large Object*).

Git n'utilise donc pas les noms des fichiers et des répertoires pour classer et stocker vos données, mais il utilise leur empreinte ou identifiant SHA-1 !

Il existe un dernier objet qui ne l'est pas vraiment. o_O C'est l'objet Tag. Le tag va représenter un commit d'une version spécifique.



## Représentation cryptographique d'un commit

Toutes les informations nécessaires pour décrire l’historique d’un projet sont stockées dans des fichiers référencés par un identifiant de 40 caractères qui ressemble à quelque chose comme ça :

8gh96c4636981e4759825791c8ea3bcf5f2278t9

Pour chacun des objets dans Git, vous trouverez cette chaîne de 40 caractères que nous appelons le **hash SHA-1.** Celui-ci représente le contenu de l'objet. Pour deux objets différents, il est donc impossible d'avoir le même nom. Cela a l'avantage que par conséquent, Git peut tout de suite reconnaître deux objets identiques. **Le commit étant un objet**, lui aussi a son empreinte SHA-1. Il est donc tout à fait possible d'appeler n'importe quel commit à n'importe quel moment grâce à cet identifiant unique.

<https://www.sha1.fr/>

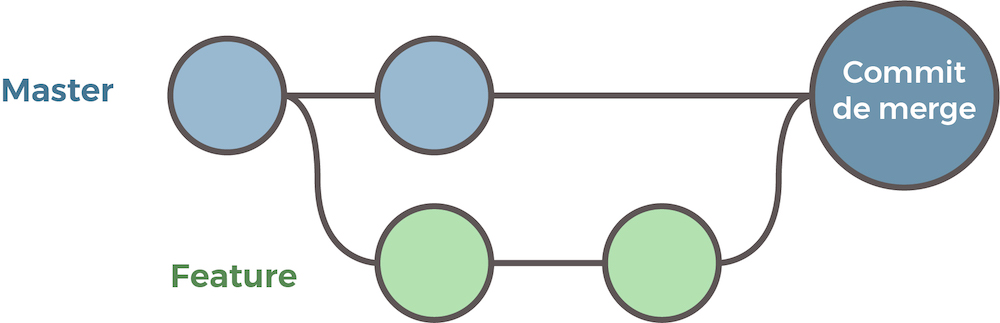
## Comment fonctionne la fusion sous Git ?

Il est très courant sous Git de vouloir fusionner le travail fait sur différentes branches. Pour cela, nous avons la fonction **Merge**. Un  **git merge**  ne devrait être utilisé que pour la récupération fonctionnelle, intégrale et finale d’une branche dans une autre, afin de préserver un graphe d’historique sémantiquement cohérent et utile, lequel représente une véritable valeur ajoutée. Comme son nom l’indique,  merge  réalise une **fusion**.  git merge  va combiner plusieurs séquences de commits en un historique unifié. Le plus souvent,   git merge  est utilisé pour combiner deux branches.  git merge  va créer un nouveau commit de merge.

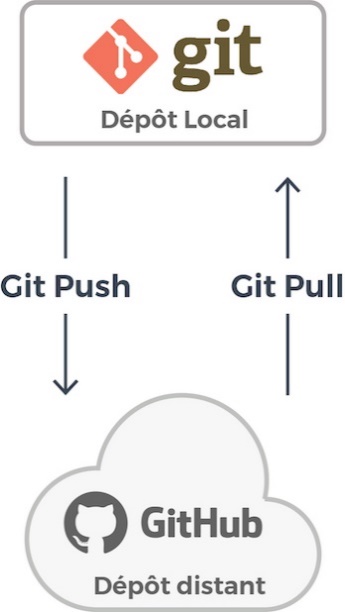
**Vous devez toujours vous assurer d'être sur la bonne branche. Pour cela, vous pouvez réaliser un  git status**

**git merge Nouvelle Fonctionnalité**

Votre branche Nouvelle fonctionnalité va être fusionnée sur la branche master en créant un nouveau commit.



## Les options Git pull/Git push

La commande Git pull permet de télécharger les modifications qui ont eu lieu sur le dépôt distant, dans le but de les rapatrier sur le dépôt local. Git pull est en réalité la fusion de deux commandes Git :  git merge  que nous venons de voir et  git fetch  que nous verrons juste après.  Git pull va créer un nouveau commit de fusion comme le fait  git merge. La commande  git pull  exécute d’abord  git fetch  qui télécharge le contenu du référentiel distant spécifié. Ensuite, un git merge  est exécuté pour fusionner les modifications du dépôt distant et créer un nouveau commit de merge en local.

**Git pull <remote>**

À l’inverse, la commande Git push permet d’envoyer des modifications que l’on a réalisées en local sur le dépôt à distance.

**Git push -u nomRepGitHub master**

## À quoi sert Git fetch ?

Git fetch, contrairement à Git pull, va aller **chercher les modifications** sur le dépôt distant mais **ne va pas les fusionner avec nos modifications locales**. Git isole le contenu récupéré en tant que contenu local existant, cela n'a absolument aucun effet sur votre travail de développement local. La commande Git fetch va récupérer toutes les données des commits effectués sur la branche courante qui n'existent pas encore dans votre version en local. **Ces données seront stockées dans le répertoire de travail local**, mais ne seront pas fusionnées avec votre branche locale. Si vous souhaitez fusionner ces données pour que votre branche soit à jour, vous devez utiliser ensuite la commande Git merge.

