GIT : Qu'est-ce qu’un gestionnaire de versions ?

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

Si vous travaillez seul, vous pourrez garder l’historique de vos modifications ou revenir à une version précédente facilement.

Si vous travaillez en équipe, plus besoin de mener votre enquête ! Le gestionnaire de versions fusionne les modifications des personnes qui travaillent simultanément sur un même fichier. Grâce à ça, vous ne risquez plus de voir votre travail supprimé par erreur !

*GIT a donc trois grandes fonctionnalités :*

* Revenir à une version précédente de votre code en cas de problème.
* Suivre l’évolution de votre code étape par étape.
* Travailler à plusieurs sans risquer de supprimer les modifications des autres collaborateurs.

Git ou GitHub ? Quelle est la différence ?

Git et GitHub sont deux choses différentes.

Git est un gestionnaire de versions. Vous l’utiliserez pour créer un dépôt local et gérer les versions de vos fichiers.

GitHub est un service en ligne qui va héberger votre dépôt. Dans ce cas, on parle de dépôt distant puisqu’il n’est pas stocké sur votre machine.

Avec Git, vous préparez votre code. Avec GitHub, vous stockez votre code.

*Repository(dépôt)* : est comme un dossier qui conserve un historique des versions et des modifications d’un projet. Il peut être local ou distant.

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

On réalise une version, que l'on va petit à petit améliorer. Ces versions sont stockées au fur et à mesure dans le dépôt local.

**Le dépôt distant :** Le dépôt distant est un peu différent. Il permet de stocker les différentes versions de votre code afin de garder un historique délocalisé, c'est-à-dire un historique hébergé sur Internet ou sur un réseau. Vous pouvez avoir plusieurs dépôts distants avec des droits différents (lecture seule, écriture, etc.).

Le dépôt distant est donc un type de dépôt indispensable si vous travaillez à plusieurs sur le même projet, puisqu’il permet de centraliser le travail de chaque développeur. C’est aussi sur le dépôt distant que toutes les modifications de tous les collaborateurs seront fusionnées.

Alors, pourquoi créer une copie locale ?

Tout simplement car votre dépôt local est un clone de votre dépôt distant. C’est sur votre dépôt local que vous ferez toutes vos modifications de code.

Commandes git :

Alors après l'installation de GIT: on utilise les commandes linux cd, mkdir pour se déplacer et créer un répertoire.

* cd Documents
* mkdir /PremierProjet --> création du projet (depot local).
* cd PremierProjet
* git config --global user.name "Boussaid Mohammed" --> associer le nom au projet
* git config --global user.email boussaidmohammed88@gmail.com --> associer l'email au projet
* git config --global color.diff auto
* git config --global color.status auto
* git config --global color.branch auto --> ces 3 dernieres commande sont utiliser pour configurer les couleurs de façons automatiques sous git.
* git config --global core.editor vim --> configurer l'éditeur á utiliser, ici c'est vim
* git config --global merge.tool vimdiff --> configurer l'outil de fusion, ici c'est vimdiff
* git config --list --> pour afficher la configuration courante du GIT
* pwd 🡪 pour savoir le dossier courant.

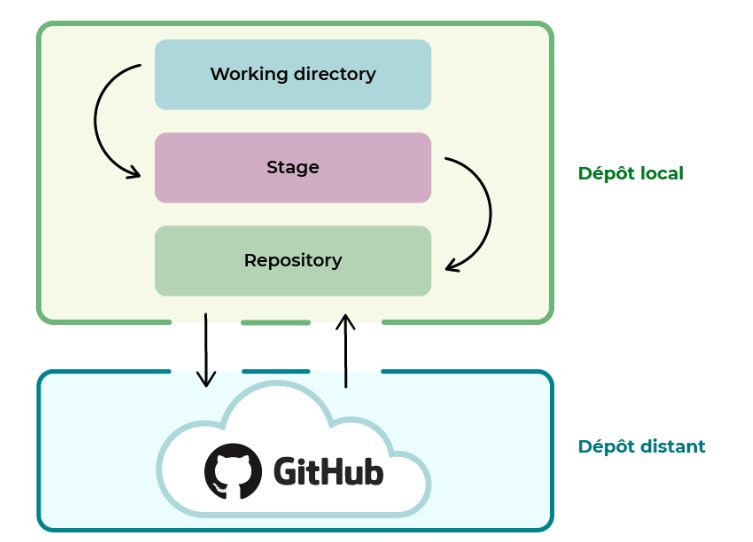
Créez votre dépôt local

Les deux méthodes pour créer un dépôt local

* Créer un dépôt local vide pour accueillir un nouveau projet : la procédure est expliquée ci-dessous ;
* Cloner un dépôt distant, c’est-à-dire recopier l’historique d’un dépôt distant en local.

La commande "git init" utilisée pour initialiser votre dépôt Git. Un dossier caché .git sera créé ,sachez que ce dossier caché contient tous les éléments non visibles de Git : la configuration, les “logs”, les branches...

Fonctionnement de GIT



Ce schéma représente le fonctionnement de Git. Il est composé de 3 zones qui forment le dépôt local, et le dépôt distant de GitHub.

Regardons plus en détail les différentes zones du dépôt local.

**Le Working directory**

Cette zone correspond au dossier du projet sur votre ordinateur.

**Le Stage ou index**

Cette zone est un intermédiaire entre le working directory et le repository. Elle représente tous les fichiers modifiés que vous souhaitez voir apparaître dans votre prochaine version de code.

**Le Repository**

Lorsque l’on crée de nouvelles versions d’un projet c’est dans cette zone qu’elles sont stockées.

Ces 3 zones sont donc présentes dans votre ordinateur, en local.

Stage un fichier ou indexer un fichier

Imaginez un projet composé de 3 fichiers : fichier1, fichier2 et fichier3.

Nous faisons une modification sur fichier1, puis une modification sur fichier2, depuis le working directory. Nous aimerions sauvegarder cette version grâce à Git, c'est-à-dire la stocker dans le repository.

Pour commencer, nous allons envoyer les fichiers modifiés (fichier1 et fichier2) du working directory vers l’index. **On dit qu’on va indexer** fichier1 et fichier2. Une fois les fichiers sont indexés, nous pouvons faire un commit pour créer une nouvelle version de notre projet, du coup enregistrer les modification sur le dépôt local .

Tout d’abord il faut créer un fichier index.html dans le working directory, avec la commande :

-touch index.html 🡪 créer un fichier index.html

-git status 🡪 fichiers verts sont indexés, fichiers rouges sont modifié mais pas indéxés.

-git add index.html 🡪 permet d’ajouté le fichier index.html dans le stage area.

-git commit -m "ajoute de fichier index.html" 🡪 permet de mettre les modifications faites dans le dépôt locale (Repository).

#### Envoyez votre commit sur le dépôt distant avec la commande git push

Créer une version de votre projet, c’est super, mais il faut désormais passer votre commit du repository à votre dépôt distant. On dit qu’il faut “pusher” votre commit.

Pour commencer, vous allez devoir “reliez” votre dépôt local au dépôt distant que vous avez créé sur GitHub précédemment. Pour cela il faut aller á votre github et dans la partie repositories, il faut avoir le lien de l’un de vous repositorys distants.

* git remote add origin https://github.com/BoussaidMohammed/OpenClassRoomsProjects.git
* git branch -M master

Ça y est ! Vous avez relié le dépôt local au dépôt distant. Vous pouvez donc envoyer des commits du repository local vers le repository distant GitHub en utilisant la commande suivante :

* git push -u origin master 🡪 -u pour le premier push.
* Git push origin master 🡪 pour toutes les pushs dès la 2ème fois.

### En résumé

* git add permet d’ajouter des fichiers dans l’index, qui est une zone intermédiaire dans laquelle stocker les fichiers modifiés.
* git commit permet de créer une nouvelle version avec les fichiers situés dans l’index.
* git commit -m permet de créer une nouvelle version et de préciser le message rattaché au commit.
* git push permet d’envoyer les modifications faites en local vers un dépôt distant.

## Appréhendez le système de branches

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.

La branche principale (main ou master) portera l’intégralité des modifications effectuées. Le but n’est donc pas de réaliser les modifications directement sur cette branche, mais de les réaliser sur d’autres branches, et après divers tests, de les intégrer sur la branche principale.

Avec Git et ses fameuses branches, pas de soucis. Vous pouvez créer une branche correspondant à l’évolution courante, et cela **sans toucher à votre application en cours de production**.

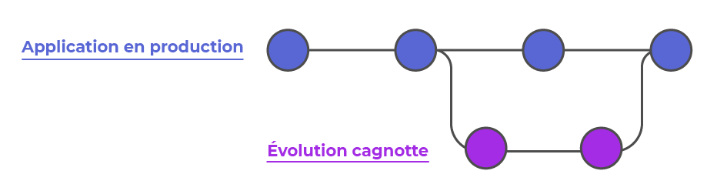
Une fois que toutes vos modifications auront été testées, vous pourrez les envoyer en production sans crainte en les intégrant à la branche main (et dans le pire des cas, revenir en arrière simplement) !

$ git branch 🡪 montre la branche sur laquelle en travaille en cours.

$ git branch cagnotte 🡪 ajoute un nouveau branch « cagnote »

$ git checkout cagnotte 🡪 permet de basculer vers le nouveau branche.

Vous pouvez désormais réaliser votre évolution sans toucher à la branche main qui abrite votre code principal fonctionnel. Vous pouvez re-basculer si besoin à tout moment sur la branche main, sans impacter les modifications de la branche Cagnotte.



Fusionnez votre travail avec la commande git merge :

$ git checkout master 🡪 il faut premièrement revenir sur la branche master

$ git merge cagnotte 🡪 et finalement fusionner la branche master avec cagnotte.

## Travaillez avec un dépôt distant

$ git clone <https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-et-GitHub.git> 🡪 permet de copier un dépôt distant en locale.

$ git remote add OC <https://github.com/OpenClassrooms-Student-Center/7162856-G-rez-Git-et-GitHub.git> 🡪 permet de donner un ALIAS au dépôt distant.

$ git pull OC main 🡪 dans le cas où un de vos amis a fait des mis-a-jour sur la branche principale et vous souhaitez les avoir localement.

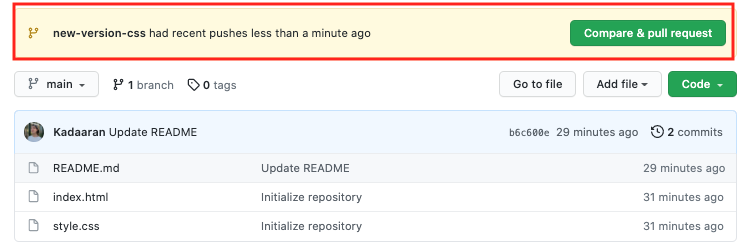
Dans les chapitres précédents, git merge nous permettait de fusionner les modifications de notre branche avec la branche principale.

Mais dans un contexte professionnel, c’est un peu plus compliqué ! 😀 Lorsque vous travaillez en équipe sur un repository, la branche principale est souvent bloquée. Vous ne pouvez pas pusher directement votre code sans qu’il soit vérifié. Vous ne pouvez donc pas fusionner vos modifications vous-même !

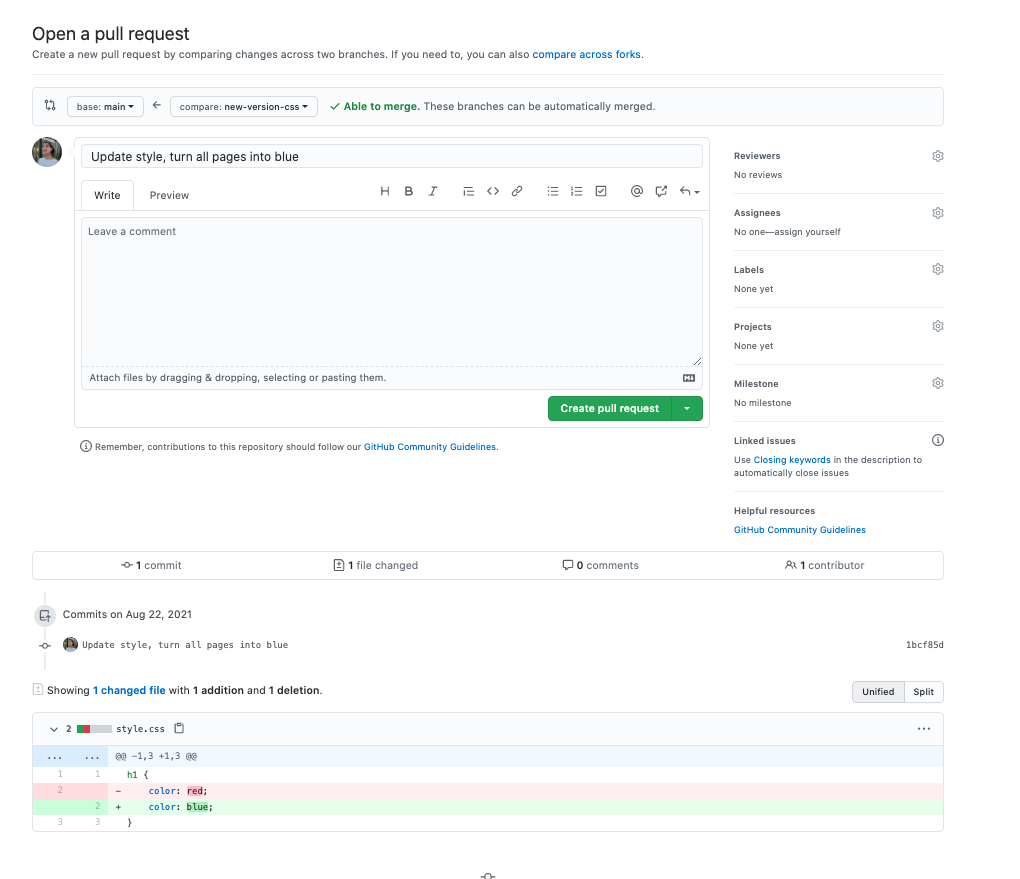
#### ****Réalisez un pull request****

Un pull **request**, ou demande de pull, en français, est une fonctionnalité de GitHub qui permet de demander aux propriétaires d’un repository l’autorisation de fusionner nos changements sur la branche principale ou toute autre branche sur laquelle on souhaite apporter nos modifications.

Au préalable il faut Créé une nouvelle branche et de l’envoyé vers le dépôt distant. Si c’est fait alors un bandeau apparaît à l’écran pour vous suggérer de créer un pull request :



Ensuite il faut remplir les raisons de vos modifications.



### En résumé

* Sur GitHub, nous pouvons récupérer l'URL d'un dépôt distant.
* git clone permet de copier en local un dépôt distant.
* git remote add permet de lier un dépôt à un "nom court", pour une plus grande facilité d’utilisation.
* git pull permet de dupliquer un dépôt GitHub en local.
* Une Pull Request permet de demander à fusionner votre code sur la branche principale.

## Pratiquez et corrigez vos erreurs sur un dépôt local

Avant de créer une branche, vous devez créer votre branche principale.

Pour cela, il vous suffit d’ajouter un fichier et de le commiter.

$ git branch -d branchTest 🡪 l’option -d permet de supprimer une nouvelle branche qui était ajoutée par erreur.

$ git branch -D brancheTest 🡪 permet même de supprimer la branche avec toutes les modifications faites.

* $ git stash 🡪 permet de revenir sur le dernier commit de la branche, et également de récupérer les modifications que vous avez rangées dans le stash
* $ git stash list 🡪 permet de voir la liste des édentifiants des stash afin de les utilisé dans la restauration dans une autre branche.
* $ git statsh apply stash@{0} 🡪 restauration du stash0 dans la branche actuelle.
* $ git log 🡪 affiche la liste des derniers commits.
* $ git reset --hard HEAD^ 🡪 HEAD^ permet de supprimer le dernier commit.
* $ git reset --hard 630efe7423d919dff53f9b54b5c59ea3c391ca1f 🡪 l’identifiant d’un commit ainsi que la commande get reset –hard nous permettre d’appliquer un commit sur la branche actuelle.
* $ git commit --amend -m "Votre nouveau message de commit" 🡪 permet de renouveler le commentaire du dernier commit.
* $ git commit --amend --no-edit 🡪 dans le cas ou vous allez oublier d’ajouter un fichier dans le dernier commit, cette commande l’ajoute facilement.

### En résumé

* git branch -d permet de supprimer une branche.
* git status permet de voir l’état des fichiers.
* git stash enregistre les modifications non indexées pour une utilisation ultérieure.
* git log affiche l'historique des commits réalisés sur la branche courante.
* git reset --hard HEAD^ permet de réinitialiser l'index et le répertoire de travail à l'état du dernier commit.
* git commit --amend permet de sélectionner le dernier commit pour y effectuer des modifications.

## Corrigez vos erreurs sur votre dépôt distant

**Vous avez pushé**des fichiers erronés.

Le problème, c'est que cette erreur concerne aussi les personnes avec lesquels vous travaillez sur le projet.

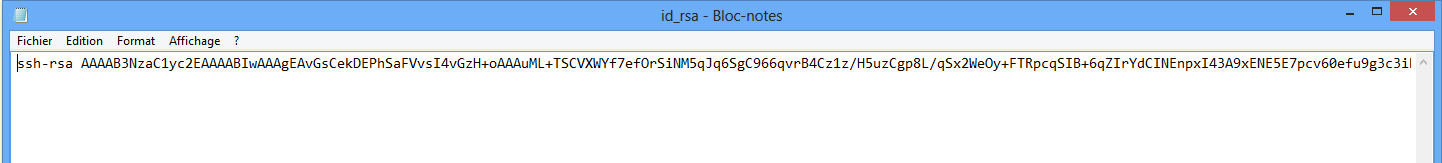
$ git revert HEAD^ 🡪 L'opération revert annule un commit en créant un nouveau commit

### L'accès à distance ne fonctionne pas

Si votre accès à distance ne fonctionne pas, cela peut être dû à un problème d’authentification de votre réseau. Pour le résoudre, il vous faut créer une paire de **clés SSH**.

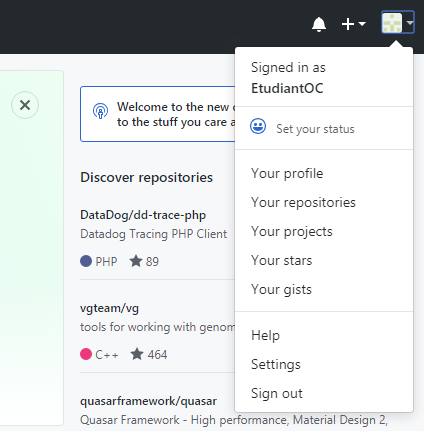
$ ssh-keygen -t rsa -b 4096 -C [Boussaidmohammed88@gmail.com](mailto:Boussaidmohammed88@gmail.com) 🡪 permet de génerer une clé ssh dans le dossier courant.

La clé id\_rsa.txt est votre **clé privée** alors que la clé id\_rsa.pub est votre**clé publique**. Ici nous allons utiliser votre clé **publique** seulement. Vous pouvez copier votre clé **publique** en l'ouvrant dans un bloc-notes.



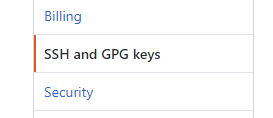
Ajoutez maintenant la clé à votre compte GitHub.

Connectez-vous à votre espace GitHub, puis allez dans l'angle droit de votre compte et cliquez sur Settings.



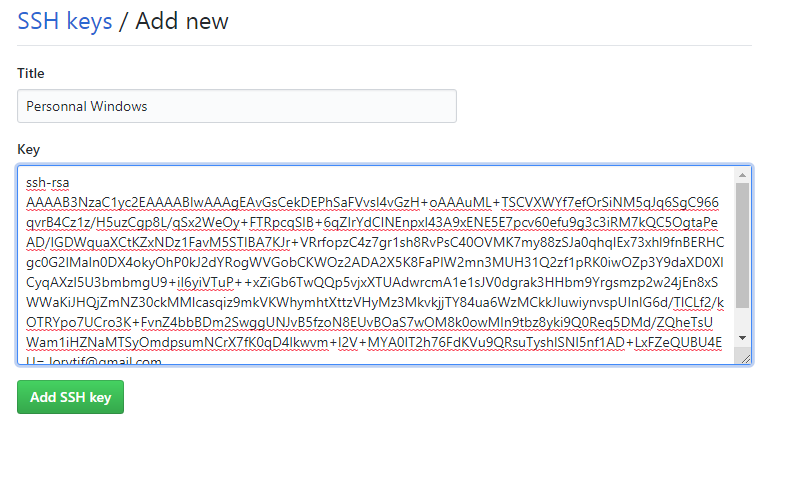
Espace "Settings" dans votre espace GitHub

Cliquez sur SSH and GPG keys :



Cliquez sur "SSH and GPG keys"

Choisissez un titre et collez votre clé SSH :



### Les trois types de réinitialisation de Git

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**.

Exécutez la commande :

git reset notreCommitCible --hard

Cette commande permet de **revenir à n'importe quel commit** mais en oubliant absolument tout ce qu'il s'est passé après ! Quand je dis tout, c'est TOUT ! Que vous ayez fait des modifications après ou d'autres commits, tout sera effacé ! C'est pourquoi il est extrêmement important de revérifier plusieurs fois avant de la lancer, vous pourriez perdre toutes vos modifications si elle est mal faite.

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

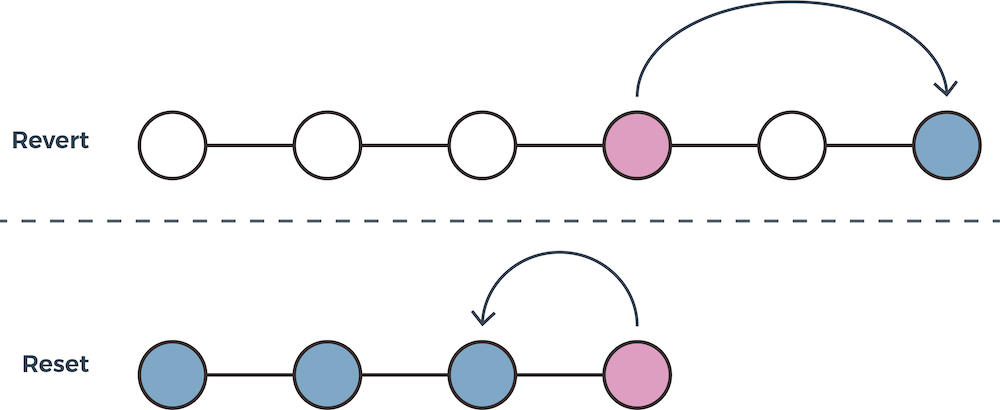
git reset HEAD~

Si rien n'est spécifié après  git reset  , par défaut il exécutera un    git reset --mixed HEAD~  .

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

Avec Git nous avons une super fonction qui va **remonter le temps**. La commande **git revert** vous permet de revenir à l'état précédent, tout en faisant un deuxième commit.  Au lieu de supprimer le commit de l'historique du projet, elle détermine comment **annuler les changements** introduits par le commit et ajoute un nouveau commit avec le contenu ainsi obtenu. Vous allez donc revenir à l'état précédent mais avec un nouveau commit.

git reset  va revenir à l'état précédent sans créer un nouveau commit, alors que  git revert  va créer un nouveau commit.



La différence entre git revert et git reset

Essayons cette super commande en faisant un premier commit que nous allons finalement ne plus vouloir. Une fois votre commit fait, écrivez la commande suivante :

git revert HEAD

Une fois votre commit "annulé", vous pouvez enlever votre fichier et réaliser de nouveau votre commit.https://user.oc-static.com/upload/2021/10/06/16334778342456_image8.png

git log

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

git reflog  va loguer les commits ainsi que toutes les autres actions que vous avez pu faire en local : vos modifications de messages, vos merges, vos resets, enfin tout, quoi  https://user.oc-static.com/upload/2021/10/06/16334784200503_image14.png. Comme    git log   ,   git reflog  affiche 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

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 https://user.oc-static.com/upload/2021/10/06/16334785104367_image31.gif .

git blame monFichier.php

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** !

Admettons que vous travailliez sur une branche "Mes évolutions", et que vous ayez déjà réalisé plusieurs commits. Votre collègue a besoin de l'une de ces évolutions pour la livrer au client, mais pas des autres. C'est dans ce cas bien précis que nous allons faire appel à    git cherry-pick  ! Cette commande va permettre de sélectionner un ou plusieurs commits grâce à leur SHA (décidément ils sont partout) et de les migrer sur la branche principale, sans pour autant fusionner toute la branche "Mes évolutions".

git cherry-pick d356940 de966d4

### En résumé

* git log affiche l'historique des commits réalisés sur la branche courante.
* git reflog est identique à git log. Cette commande affiche également toutes les actions réalisées en local.
* git checkout un\_identifiant\_SHA-1 permet de revenir à une action donnée.
* git blame permet de savoir qui a réalisé telle modification dans un fichier, à quelle date, ligne par ligne.
* git cherry-pick un\_identifiant\_SHA-1 un\_autre\_identifiant\_SHA-1 permet de sélectionner un commit et de l'appliquer sur la branche actuelle.

