# C:\Users\Dheryo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Git-Logo-2Color.pngEXPLICATIONS GIT

Tu trouveras ci-dessous quelques explications sur GIT suite à ta demande :

[EXPLICATIONS GIT 1](#_Toc500158373)

[1 - Qu'est-ce qu'un commit ? 2](#_Toc500158374)

[2 - À quoi sert la commande git log ? 3](#_Toc500158375)

[3 - Qu'est-ce qu'une branche ? 5](#_Toc500158376)

[4 - Aller plus loin 7](#_Toc500158377)

J’espère que mes explications t’aideront.

## 1 - Qu'est-ce qu'un commit ?

Un commit c'est comme une photo de l'état de ton projet.

Quand tu as fini de modifier une partie de ton projet (correction d'une fonction par exemple), tu fais une 'photo' (un commit) qui garde une image de l'état actuel du projet.

N'oublie pas d'indiquer un message explicite qui correspond à tes modifications ("Correction de la fonction 'nom\_de\_la\_fonction'" par exemple).

C'est important pour t'y retrouver plus tard et/ou travailler à plusieurs sur le projet.

Ensuite, tu peux continuer à travailler sur ton projet et consulter à tout moment les différents commits ou revenir sur l’un d’eux. C’est très pratique pour organiser ton travail, revenir facilement en arrière si besoin et pour travailler à plusieurs.

Fais autant de commits que nécessaire mais pas pour rien histoire de ne pas alourdir la liste de commits inutilement. J'insiste sur le fait de pouvoir s'y retrouver plus tard.

C'est comme quand tu as appris à respecter des conventions d'écriture dans tes programmes et à les commenter clairement.

L’intérêt de travailler ainsi devrait te paraître encore plus clair après avoir abordé les points suivants.

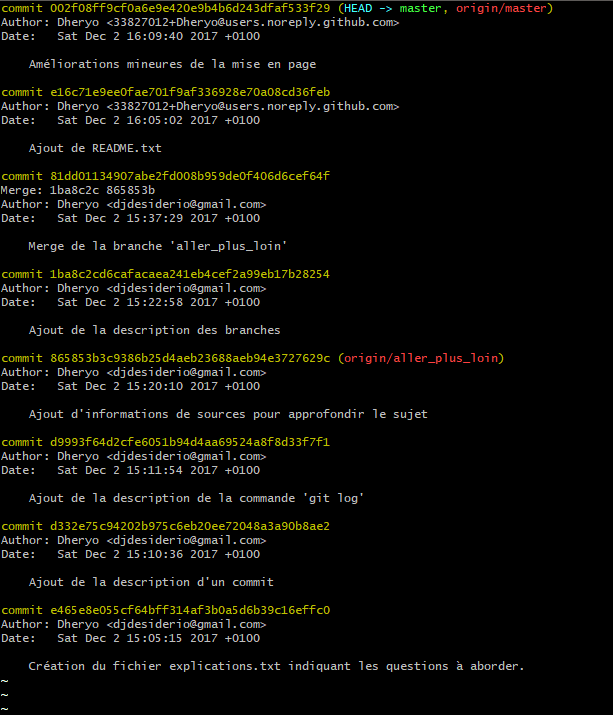
Pour rappel, tu peux faire un commit avec la commande :

'git commit -m "Explication simple et claire des modifications" '

## 2 - À quoi sert la commande git log ?

La commande ‘git log’ sert simplement à afficher la liste de tous les commits effectués sur la branche active de ton projet (je reviens sur les branches après puisque c'est ta dernière question).

Voici un extrait de log ci-dessous :



Les commits sont affichés dans l'ordre du plus récent au plus vieux.

Les informations données sont :

- le SHA (son identifiant unique qui te sert notamment à revenir sur ce commit), c'est la suite hexadécimale à rallonge,

- le nom de l'auteur pour suivre qui a fait tel ou tel commit,

- la date et l'heure à laquelle le commit a été effectué,

- et le fameux message si important qui permet de savoir facilement à quoi a servi le commit, d'où encore une fois l'intérêt d'un message clair.

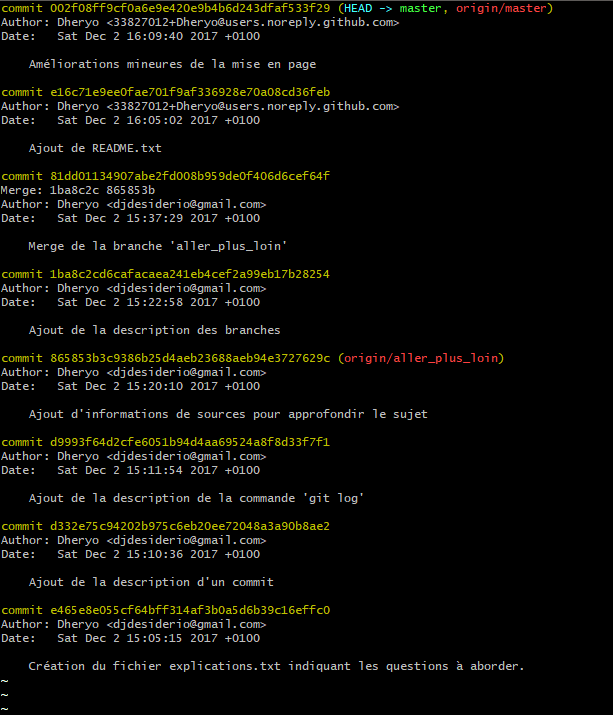
Certaines lignes comportent également des informations entre parenthèses.

L’une d’elle indique : « HEAD -> master » ce qui signifie que l’état actuel de ton projet est sur ce commit. Tu me diras : ‘Normal, c’est le dernier !’ .

Mais on peut revenir en arrière à tout moment, le « HEAD -> master » n’est donc pas obligatoirement sur le dernier commit.

Il y a également des informations en rouge relatives au repository GitHub sur lequel j’ai envoyé le projet. Je ne vais pas rentrer dans les détails là-dessus pour l’instant.

En résumé, si on reprend un commit sur le log :



**Correspondance**

**repository**

**Position**

**projet**

**SHA unique du commit**

**Message de**

**description**

**Horodatage**

**Auteur**

La commande pour afficher le log, tu la connais, c'est juste :

'git log'

## 3 - Qu'est-ce qu'une branche ?

Enfin, les branches. Il me semble qu'un exemple d'utilisation des branches sera plus parlant.

Admettons que tu développes un jeu solo. En cours de développement, tu te dis qu'il faudrait essayer d'ajouter un mode multi et tu me demandes de t'aider sur cette partie pour gagner du temps. Plutôt que travailler sur la même version du projet, je vais créer une nouvelle branche.

A partir de là, les commits que je vais faire sur ma branche que j'aurais par exemple sobrement nommée "multi" et les commits que tu feras sur la branche par défaut qui s'appelle "master" seront indépendants.

Toi tu auras un projet avec tous tes commits et moi j'aurai une version parrallèle du projet avec les commits que tu avais effectués avant la création de la branche "multi" suivis de mes commits à moi sans que les deux projets parallèles interfèrent.

A la fin, nous pourrons décider de 'merger' ma branche au "master" (c’est-à-dire les fusionner) ou de la supprimer si tu décides que le mode multi développé ne s'intègre pas dans ton jeu.

On peut aussi laisser la branche "multi" de côté en attendant de décider si elle devra être intégrée ou non.

La commande : 'git branch' permet d'afficher la liste des branches existantes et

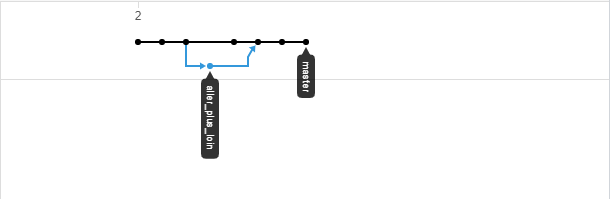
'git branch "nom\_de\_la\_branche" ' permet d'en créer une nouvelle.

Pour te déplacer d'une branche à l'autre c'est comme pour se déplacer d'un commit à l'autre avec la commande :

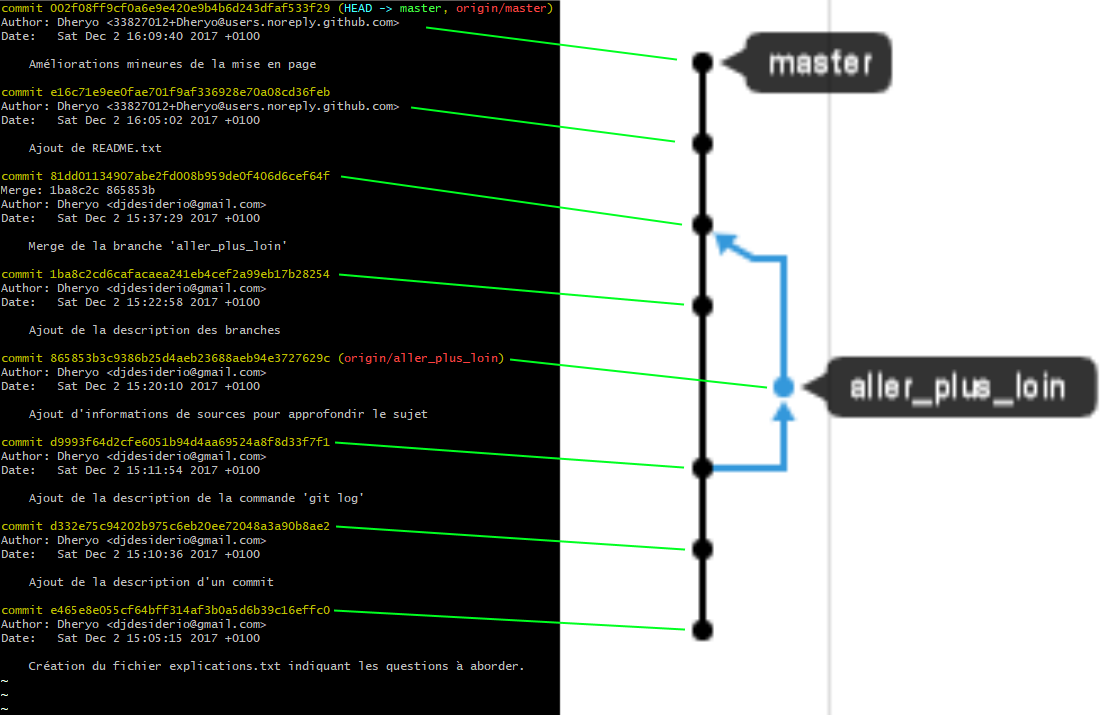
'git checkout "nom\_de\_la branche" '

Ci-dessous une représentation du même projet que celui dont j’ai mis le log plus haut.

Elle a été générée sur GitHub et reprend la branche principale « master » en noir et une seconde branche « aller\_plus\_loin » en bleu.



On peut ainsi corréler les deux images entre elles de la façon suivante :



On voit ainsi clairement :

* que chaque point correspond à un log,
* qu’après le 3ème commit dans l’ordre chronologique, c’est-à-dire en partant du bas, a été créée une nouvelle branche nommée « aller\_plus\_loin »,
* que le 4ème commit du log a été fait dans la branche « aller\_plus\_loin »,
* que le commit suivant a été effectué sur la branche « master ». A ce stade, chaque branche contient 4 commits, mais le dernier commit de chaque branche est différent,
* que le commit d’après consiste à ré-unifier les deux branches, ce qu’on appelle un « merge »
* que les informations en rouge que j’avais laissées de côté dans l’explication des logs correspondent à l’état de chaque branche dans le repo GitHub, le nom par défaut du repo GitHub étant « origin ».

Détail important, dans mon exemple les deux branches ont servi à modifier différemment le même fichier avant de fusionner les modifications. C’est volontaire car je voulais tester une résolution de conflit. En effet, GIT trouve des modifications différentes au même endroit du fichier. Il n’est donc pas capable d’effectuer le merge automatiquement et réclame une action manuelle sur le fichier.

Attention donc à l’organisation des tâches dans un projet afin de limiter les conflits dus par exemple à deux personnes qui modifient une même partie d’un fichier dans 2 branches différentes.

## 4 - Aller plus loin

Pour plus d'informations, deux grands classiques :

- consulter la documentation :

<https://git-scm.com/documentation>

- taper le nom d'une commande avec l'argument -h pour afficher l'aide dans ta console.

Pense à l'ami [Google](https://www.google.fr) et au site [Openclassrooms](http://openclassrooms.com) sur lequel je me forme.

J'ai également créé un repository sur Github pour illustrer ce que je viens de t'expliquer.

Il se trouve à l'adresse suivante :

<https://github.com/Dheryo/explications_git>

Bonne continuation !