Cégep de Saint-Hyacinthe
Département d'informatique

Programmation client-serveur

420-2RP-HY

Notes de cours

Gestion de versions de programmes 1

Enseignante
Giovana Velarde

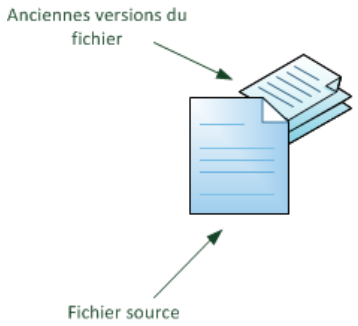
Hiver 2022

Table des matières

Gestion de versions de programmes

Un logiciel de gestion de versions (*Version Control System*) est utilisé principalement par les développeurs pour gérer des codes sources, car il est capable de suivre l'évolution d'un **fichier texte** ligne de code par ligne de code et garde les anciennes versions de chacun d'eux sans rien écraser. Cela permet de retrouver les différentes versions d'un fichier ou d'un lot de fichiers connexes et ainsi éviter des problèmes tel quels : de la perte du code, des bugs après des modifications, etc. Il permet de retrouver sans problème la version qui fonctionnait avant.

Ce logiciel est fortement conseillé pour gérer un projet informatique. Il est devenu indispensable lorsqu'on travaille à plusieurs sur un même projet et donc sur le même code source. Même si on travaille seul, il offre de nombreux avantages, comme la conservation d'un historique de chaque modification des fichiers par exemple.



En plus d'être un outil de *backup* (sauvegarde), il propose de nombreuses fonctionnalités tout au long de l'évolution de votre projet informatique :

- Il retient qui a effectué chaque modification de chaque fichier et pourquoi.
- Si deux personnes travaillent simultanément sur un même fichier, il est capable d'assembler (de fusionner) leurs modifications et d'éviter que le travail d'une de ces personnes ne soit écrasé.

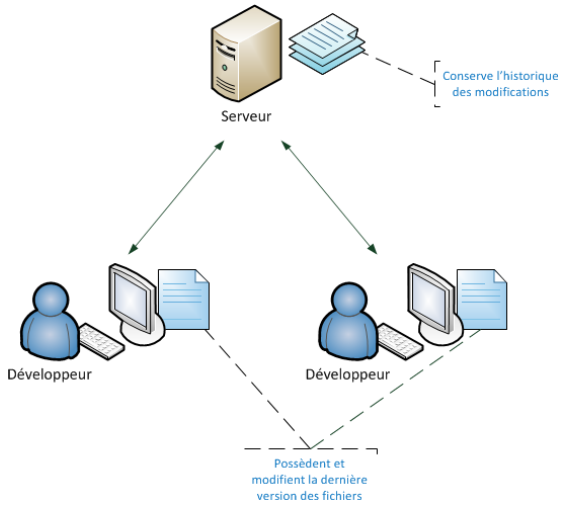
Ce logiciel a deux utilités principales :

- **Suivre l'évolution d'un code source**, pour retenir les modifications effectuées sur chaque fichier et être ainsi capable de revenir en arrière en cas de problème parce que le logiciel conserve une copie de tous les changements produits.
- **Travailler à plusieurs**. Si deux personnes modifient un même fichier en même temps, deux copies seront sauvegardées et, plus tard, leurs modifications peuvent être fusionnées sans perte d'information.

1 Logiciel centralisé vs distribué

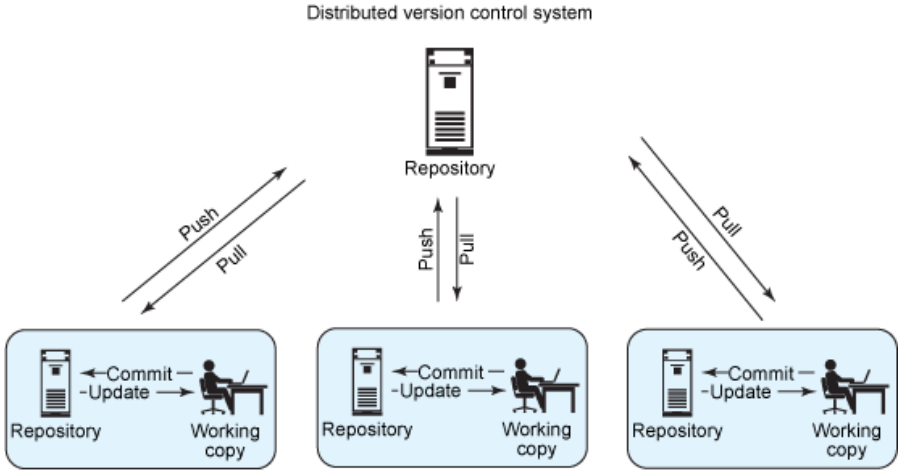
Il existe deux types principaux de logiciel de gestion de versions.

- **Le logiciel centralisé** : 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.



Un logiciel de gestion de versions centralisé

- **Le logiciel distribué** : Chaque développeur possède un dépôt local qui est un clone du dépôt central. Il y enregistre l'historique de l'évolution local de chacun des fichiers. Il envoie l'historique des modifications vers le dépôt central périodiquement.



Un logiciel de gestion de versions distribué

Il existe de nombreux logiciels de gestion de versions gratuits (comme *SVN*, *Mercurial*, *Bazaar* et *Git*) et propriétaires (comme *Perforce*, *BitKeeper*, *Visual SourceSafe*, etc.)

2 Outils

2.1 Git

Git est un des plus puissants logiciels de gestion de versions. Caractéristiques :

- Distribué
- Très rapide
- Travaille par branches (versions parallèles d'un même projet) de façon très flexible
- Assez complexe, il faut un certain temps d'adaptation pour bien le comprendre et le manipuler
- Multiplateforme (Linux, Windows, etc.)

2.2 GitHub

GitHub est une plateforme Web de "codes". Il fournit le **serveur web** où les développeurs qui utilisent Git se rencontrent pour le travail collaboratif. C'est une sorte de réseau social pour développeurs. Vous y pouvez regarder les projets évoluer et participer à l'un d'entre eux. 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. Par exemple : un dépôt de exemples JavaScript se trouve dans https://github.com/trekhleb/javascript-algorithms.

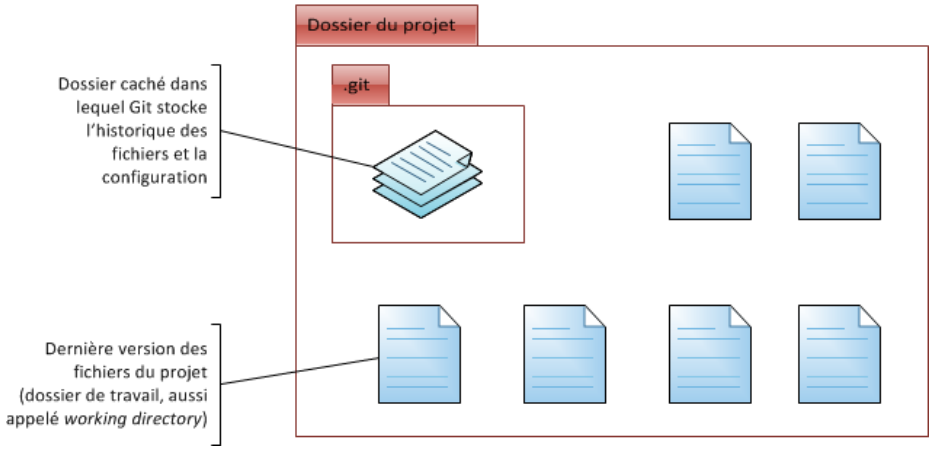
3 Fonctionnement de Git

3.1 Dépôt Git

Un dépôt Git représente les **fichiers d'un projet** ainsi que leur **historique**.

Il y deux options pour commencer à travailler avec Git :

- **CRÉER UN NOUVEAU DÉPÔT VIDE**. Il faut créer le dossier du projet sur le disque local (s'il n'existe déjà) et s'y placer à l'intérieur. Ensuite, on doit initialiser le dépôt Git. Un dossier caché « `.git` » est créé dans la racine du projet. Il contient l'historique des modifications des fichiers et la configuration de Git pour ce projet (fichier `config`).
- **CLONER UN DÉPÔT EXISTANT**. Consiste à récupérer tout l'historique de changements et tous les codes source d'un projet pour pouvoir travailler dessus. On trouve dans GitHub beaucoup de projets open-source que on peut cloner. Il faudra s'identifier en entrant un mot de passe (sauf si le dépôt a autorisé l'accès publique). Mis à part le dossier « `.git` » caché, on retrouve dans le dossier du projet tous les fichiers du projet dans leur dernière version.



Chaque développeur qui travaille sur le même projet possède dépôt local dans son ordinateur de travail. Ce dépôt local est une copie du dépôt web central. Il y a plusieurs commandes dans Git pour gérer les versions d'un projet. La liste est dans la documentation officiel https://git-scm.com/docs.

3.2 Niveaux de travail du projet local

En résumé, Git fait des *commits*. C'est-à-dire, des instantanés (*snapshots*) d'un projet pour le versionner. Pour y arriver, Git utilise trois différents niveaux de d'organisation :

ESPACE DE TRAVAIL (*WORKING DIRECTORY*) – ÉTAT MODIFIÉ

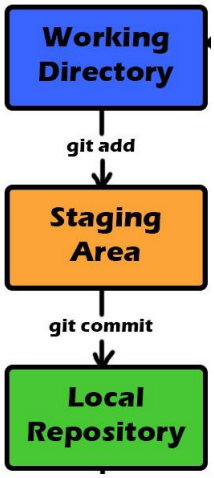
C'est là où le dépôt Git est initialisé et que les fichiers sont placés. C'est dans cette zone que on touche à tous les fichiers pendant qu'on travaille localement en ajoutant, modifiant ou supprimant du code ou des fichiers. Une fois que on veut garder une version du projet dans l'historique de versions, on doit l'indexer (`stage`) pour passer à la zone de transit.

ZONE DE TRANSIT (*STAGING AREA* OU *INDEX*) – ÉTAT INDEXÉ

C'est l'endroit où sont listées les modifications apportées dans votre environnement de travail local. Une fois que on a désigné tout ce que on voulait modifier, les modifications pourront être envoyées au dépôt local (`commit`). Seulement les fichiers indexés seront envoyés.

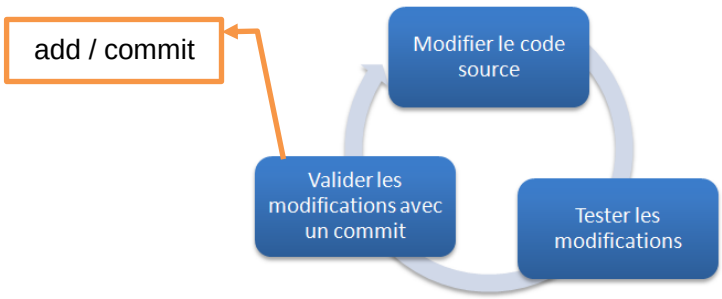
DÉPÔT LOCAL (*LOCAL REPOSITORY*) – ÉTAT VALIDÉ

C'est dans ce dépôt local que les fameux instantanés du projet sont versionnés et stockés. C'est important de comprendre qu'**une référence de version** est créée pour chaque validation (`commit`) faite. Chaque validation est une version du projet unique qui est placée dans le dépôt local et que on pourra la consulter, la comparer ou revenir au besoin.

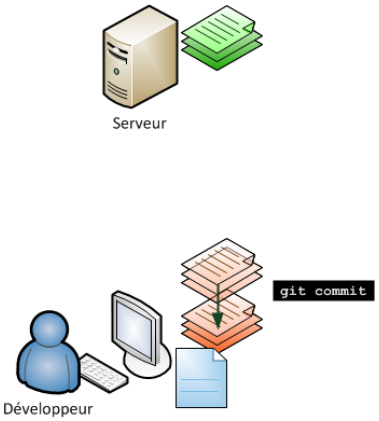


Dans la programmation on suit les étapes suivantes :

1. Modifier le code source
2. Tester votre programme pour vérifier si cela fonctionne
3. Indexer les modifications (stage) et faire une validation (commit) pour « enregistrer localement » les changements et les faire connaître à Git
4. Recommencer à partir de l'étape 1 pour une autre modification.



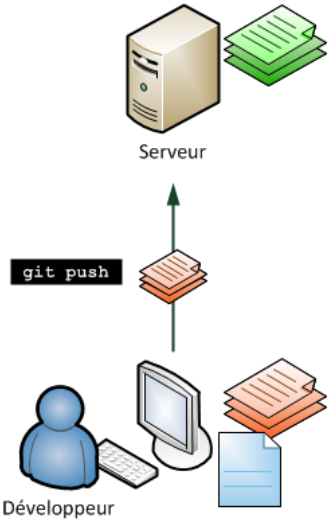
Un *commit* représente donc un ensemble de changements qui permet soit de régler un bug, soit d'ajouter une fonctionnalité. Une fois le message de *commit* enregistré, Git va officiellement sauvegarder vos changements dans un historique de versions (ligne du temps). Il ajoute donc cela à la liste des changements (instantanés) qu'il connaît du projet. Un *commit* avec Git est local seulement.



Les *commits* sont là pour « valider » l'avancement de votre projet localement. N'en faites pas un pour chaque ligne de code modifiée, mais n'attendez pas d'avoir fait 50 modifications différentes non plus !

3.3 Envoyer nos *commits* au dépôt central distant

Une fois qu'on a fini une étape du projet et on l'a versionné, on va pouvoir le partager en le poussant sur le dépôt central distant (`push`) qui envoie nos nouvelles modifications (*commits*) sur le serveur central. Le dépôt distant est appelé dépôt central car il sert de point de rencontre entre les développeurs du projet.



Avant d'envoyer vos *commits* au dépôt distant, vérifiez que tout est conforme et qu'il n'y a pas d'erreur. Il est encore temps de corriger ces *commits*, mais une fois envoyés, il sera trop tard !

3.4 Consultation des logs et corrections

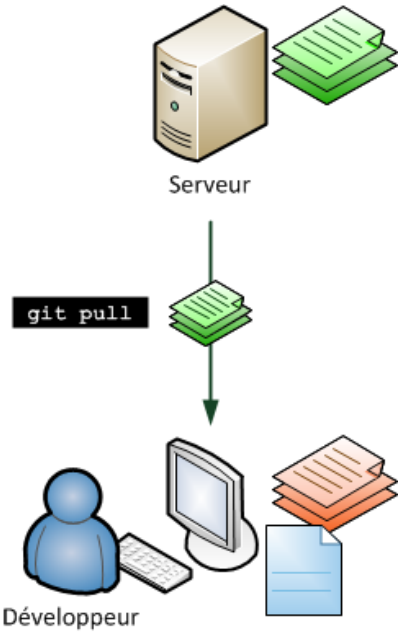
Il est possible à tout moment de consulter l'historique des *commits* (logs). On peut ainsi retrouver tout ce qui a été changé depuis les débuts du projet. Chaque commit est **numéroté** grâce à un long numéro hexadécimal qui permet de les identifier.

Par exemple : `12328a1bcbf231da8eaf942f8d68c7dc0c7c4f38`.

On peut aussi annuler des modifications indexées (*staged*) ou même des validations (*commited*) mais ce n'est pas une bonne pratique.

3.5 Télécharger les nouveautés du projet collaboratif

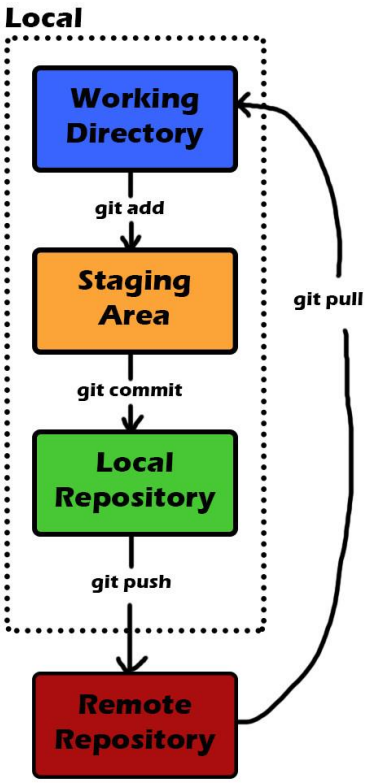
On peut télécharger les modifications effectuées le dépôt central (serveur distant) par d'autres développeurs et les intégrer dans le dépôt local (`pull`)



Deux cas sont possibles :

- Si on n'a effectué aucune modification depuis le dernier *pull*, la mise à jour est simple.
- Si on a fait des *commits* localement en même temps que d'autres programmeurs (déjà poussés dans le serveur). Leurs modifications seront alors fusionnées aux vôtres automatiquement. Si deux personnes modifient en même temps deux endroits distincts d'un même fichier, les changements sont intelligemment fusionnés par Git. Mais si deux personnes modifient la même zone de code en même temps (les mêmes lignes de code), Git signalera qu'il y a un **conflit** car il ne peut décider quelle modification doit être conservée et quelle ignorée. Il indique alors le nom des fichiers en conflit. Les symboles « <<<<<<<<< » délimiteront nos changements et ceux des autres programmeurs. On doit supprimer ces symboles et garder uniquement les changements nécessaires, puis faire un nouveau commit pour enregistrer le tout.

Les commandes les plus souvent utilisées sont l'indexation (*stage*/*add*) et la validation (*commit*) De temps en temps on utilise aussi les commandes pousser (*push*) et tirer (*pull*).



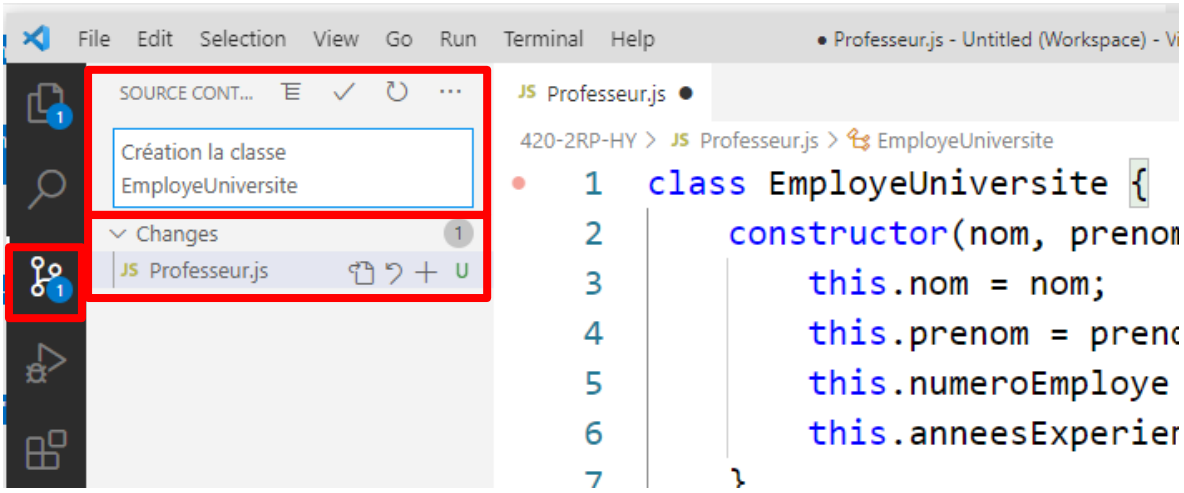
Le changement vers le serveur doit être de type *fast-forward* car le serveur ne peut régler les conflits à notre place s'il y en a. Personne ne doit avoir fait un *push* avant vous depuis votre dernier *pull*. Le mieux est de s'assurer que vous êtes à jour en faisant un pull avant de faire un *push*.

Il est recommandé de faire régulièrement des *commits*, mais pas des *push*. On ne devrait faire un *push* qu'une fois de temps en temps (pas plus d'une fois par jour en général). Éviter de faire systématiquement un *push* après chaque *commit*. Parce qu'un push est irréversible. Une fois que nos *commits* sont publiés, il deviendra impossible de les supprimer ou de modifier le message de *commit* ! On doit bien réfléchir avant de faire un *push*. (Il y a quand même un truc pour annuler un *commit* publié sur le serveur[1]).

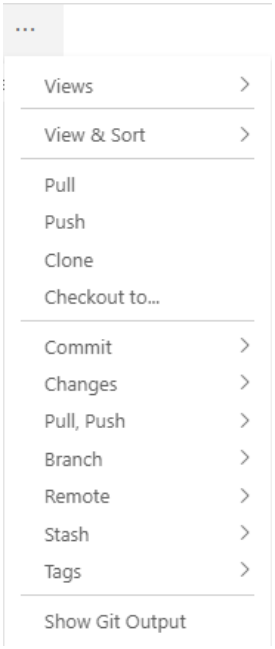
[1] https://openclassrooms.com/fr/courses/1233741-gerez-vos-codes-source-avec-git#/id/r-1234681

4 Git dans Visual Studio Code

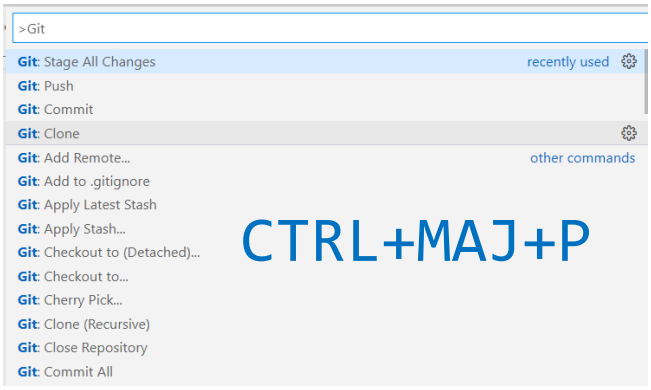
Nous pouvons utiliser les commandes de base Git dans l'environnement graphique de Visual Studio Code.



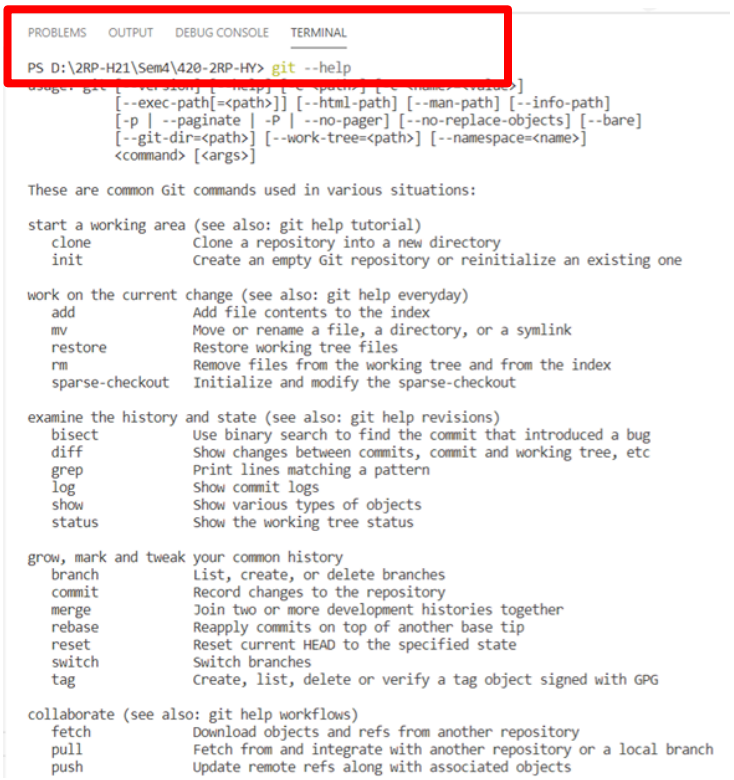
Panneau de control de versions



Menu rapide



Menu de commandes



Terminal

Exemple de travail collaboratif avec Git

5 Références

git. (2021, février 12). *Git reference*. Récupéré sur git Documentation: https://git-scm.com/docs

Visual Studio Code. (2021, janvier 1). *Git version control in VS Code.* Récupéré sur Visual Studio Code Docs : https://code.visualstudio.com/docs/introvideos/versioncontrol

Nebra, Mahieu. (2020, octobre 10). *Gérez vos codes source avec Git*. Récupéré sur OpenClassrooms: https://openclassrooms.com/fr/courses/1233741-gerez-vos-codes-source-avec-git

jesuisundev. (2020, juin 29). *Comprendre Git en 7 minutes*. Récupéré sur Je suis un dev: https://www.jesuisundev.com/comprendre-git-en-7-minutes/

Lesieur, Bruno. (2017, mai 30). *Et si vous compreniez enfin Git et GitHub?*. Récupéré sur Quelques #ID et beaucoup de .CLASS: https://blog.lesieur.name/comprendre-et-utiliser-git-avec-vos-projets/