Git & GitHub

Git, en quelques mots

Git est un outil collaboratif de gestion de version (*versioning*). La commande git permet d'utiliser Git en ligne de commande, c'est donc un outil que chaque membre d'un projet doit installer en local (sur sa machine).

Git permet de faire de la gestion de versions dans tout type de projet : pour gérer du code bien sûr, mais aussi des documents PDF, des images, etc. Concrètement, il devient possible de conserver un historique des versions successives de n'importe quels fichiers, de comparer ces versions, de revenir à une version plus ancienne en cas de pépin, etc. Ce n'est donc pas strictement un outil de développeur !

Par ailleurs, Git permet de faciliter la *collaboration* entre plusieurs utilisateurs qui travaillent sur une même base de fichiers, sans risquer d'écraser ou de perdre son travail ou celui des collègues. Pour faciliter la communication autour du projet, les bons vieux emails fonctionnent, mais il existe aujourd'hui des sites et de services complémentaires à Git. Le site GitHub est l'un d'entre eux.

Comment ça marche ?

Avec Git, quand on veut sauvegarder l'état courant de nos fichiers, on désigne quels fichiers prendre en compte pour la sauvegarde (*git add*), et on valider l'enregistrement (*git commit*) des modifications. On crée ce faisant 1 « commit », qui représente la différence entre 2 versions successives du projet. On va collectionner les commits successifs, au fur et à mesure de l'évolution du projet. Cette suite de commits constitue l'historique du projet.

Si on travaille avec d'autres personnes sur un projet géré par Git, on peut ensuite envisager de partager, fusionner ou retravailler les commits : ajouts/modifications/suppressions de fichiers, par exemple. Dans ce cas-là, on aura tendance à utiliser un serveur central auquel tout le monde aura accès pour partager l'historique du projet (*git push* vers le serveur central).

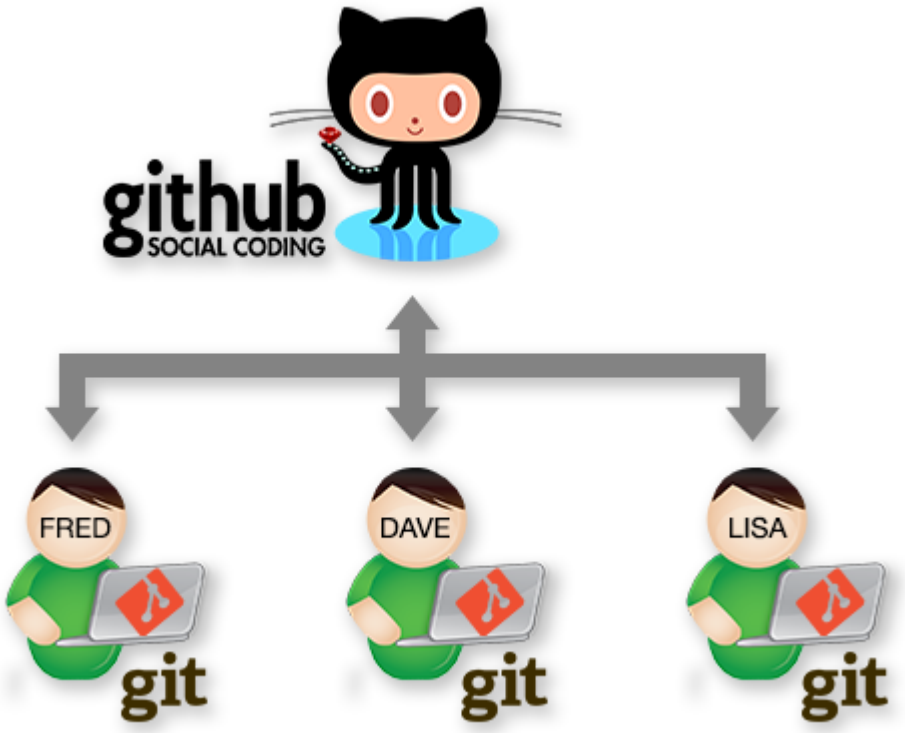
Différences entre Git & GitHub

Attention à ne pas confondre Git et GitHub :

- Git (commande git) est l'outil *open source* de gestion de version, qu'on installe sur sa machine pou coder « localement » ;
- GitHub (github.com) est une plateforme de services & un réseau social — GitHub agit comme serveur central, permettant de partager son code dans un dépôt Git centralisé et partagé sur internet, mais également de communiquer avec d'autres développeurs par l'intermédiaire de commentaires, d'*issues*, etc.

Principe général : 1 « projet Git » ===> 1 repo de code géré localement (sur sa machine) avec Git, et *éventuellement* hébergé/sauvegardé dans 1 repo distant sur github.com. Le projet est dans ce dernier cas dupliqué dans *deux* repos git distincts.

GitHub est probablement *LE* réseau social de développeurs le plus populaire aujourd'hui, mais certainement pas le seul. Il existe également Gitlab, Bitbucket, etc.



Créer une clé SSH pour GitHub

Avant toute chose, pour utiliser Git et GitHub à leur pleins potentiels, on va créer une clé dite *SSH*. Cette clé est une carte d'identité nous permettant de nous authentifier auprès de GitHub, notamment pour accéder aux repos privés, signer nos commits, etc.

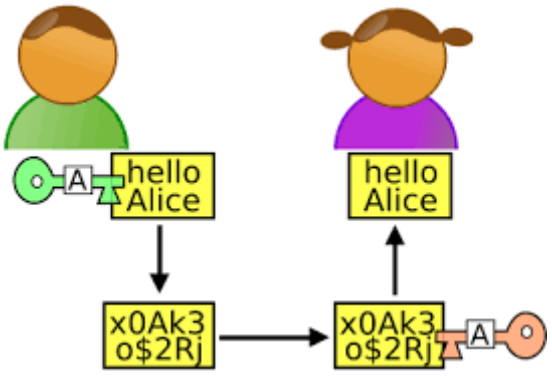
Création de la clé

```
# Attention à bien remplacer l'email par le votre ;)
ssh-keygen -t ed25519 -C "votre-email@exemple.fr"
```

Il vous sera demandé d'inventer une *passphrase*, c'est-à-dire un mot de passe un peu costaud (qui peut carrément être une phrase, avec des espaces, des accents et tout ! Cette *passphrase* n'est pas strictement obligatoire (elle peut être vide…), mais il est **fortement** recommandé d'en choisir une. Par contre, il faut la retenir par cœur, si elle est perdue, la clé SSH est bonne à jeter !

Une clé SSH se compose de deux parties, si bien qu'à l'issue de la commande, dans le téléporteur, vous obtenez deux choses :

- une clé *privée* dans /home/mint/.ssh/id_ed25519 — pour protégér du contenu, à garder pour soi !
- une clé *publique* dans /home/mint/.ssh/id_ed25519.pub — elle est capable de lire du contenu protégé par la clé privé



Les informations (historique git…) partagées entre votre machine et les serveurs de GitHub seront chiffrées, grâce à la clé privée, avant de partir de chez vous ; puis décryptées une fois arrivées sur GitHub, si celui-ci possède la clé publique.

Ajout de la clé publique sur GitHub

Vous allez donc copier le contenu de la clé *publique* sur GitHub. Vous pouvez regarder le contenu de la clé publique, par curiosité :

```
# Pour récupérer le contenu de notre clé publique
cat ~/.ssh/id_ed25519.pub
```

Copiez ce contenu, et allez le coller dans votre compte GitHub :

```
Settings > SSH and GPG keys > New SSH key > Coller le contenu de la clé et valider
```

À noter : la clé SSH publique seule ne permet pas de chiffrer du contenu, si bien que GitHub ou tout autre service auquel vous auriez donné votre clé publique ne pourra pas vous *envoyer* des informations protégées ; uniquement en décrypter. La clé SSH sert essentiellement pour vous authentifier, prouver que vous êtes la personne que vous annoncez être. Pour protéger l'échange d'information en lui-même, une seconde couche de chiffrement est ajoutée, *via* le protocole HTTPS (S comme *Secure*).

Pour en savoir plus : https://help.github.com/articles/connecting-to-github-with-ssh/

Pour que Git utilise automatiquement la clé SSH pour authentifier les commandes git ..., il *faut* utiliser des URLs avec le protocole SSH plutôt que HTTPS. À nouveau, pour en savoir plus : https://help.github.com/articles/why-is-git-always-asking-for-my-password/

Activation de la clé SSH en local

Pour que la clé SSH soit utilisable, et aussi pour éviter d'avoir à donner sa *passphrase* à chaque utilisation, il faut ajouter la clé privée à un « trousseau de clé » (programme ssh-agent):

```
eval "$(ssh-agent -s)" # pour lancer ssh-agent de façon sécurisée
ssh-add ~/.ssh/id_ed25519 # pour activer la clé SSH
```

Si vous oubliez cette étape, vous aurez des erreurs du type « Permission denied (publickey) » lors de l'utilisation de Git & GitHub.

C'est prêt !

Vérifier la connexion

Comme indiqué dans la documentation de GitHub, il est possible à ce stade de vérifier si les précédentes étapes ont bien été réalisées.

Pour cela, on va tenter une connexion SSH à github.com, en saisissant dans le terminal :

```
ssh -T git@github.com
```

Si un warning apparaît, vous pouvez vérifier la correspondance de la *fingerprint* avec celles de GitHub puis saisir *yes* si cela correspond.

Enfin, si le terminal vous répond avec votre pseudo GitHub, c'est que la connexion fonctionne. Sinon, c'est qu'il y a un problème, il faut revoir les étapes précédentes.

```
> Hi username! You've successfully authenticated, but GitHub does not provide shell access.
```

Configuration locale de Git

Git peut être configuré très précisément. Voici quelques réglages utiles à mettre en place :

Le nom affiché dans les commits :

```
# N'oubliez pas de changer le nom par le votre… ;)
git config --global user.name "Bozo Le Clown"
```

L'email associé au commit (conseil : le même que celui du compte GitHub) :

```
# N'oubliez pas de changer l'email par le votre… ;)
git config --global user.email "bozo.leclown@oclock.io"
```

Choix de l'éditeur de texte utilisé pour écrire les messages de commit :

```
git config --global core.editor code # ou nano pour les barbus, etc.
```

Activation des couleurs dans le résultat des commandes Git :

```
git config --global color.ui true
```

Vérifier la configuration

Pour vérifier le tout : git config -l. Cette commande va afficher tous les réglages actifs de git.

Créer facilement la clé SSH et la configuration Git

Le script ci-dessous t'aide à le faire sans te prendre la tête. Il te suffit de suivre les instructions fournies en commentaires au début du script.

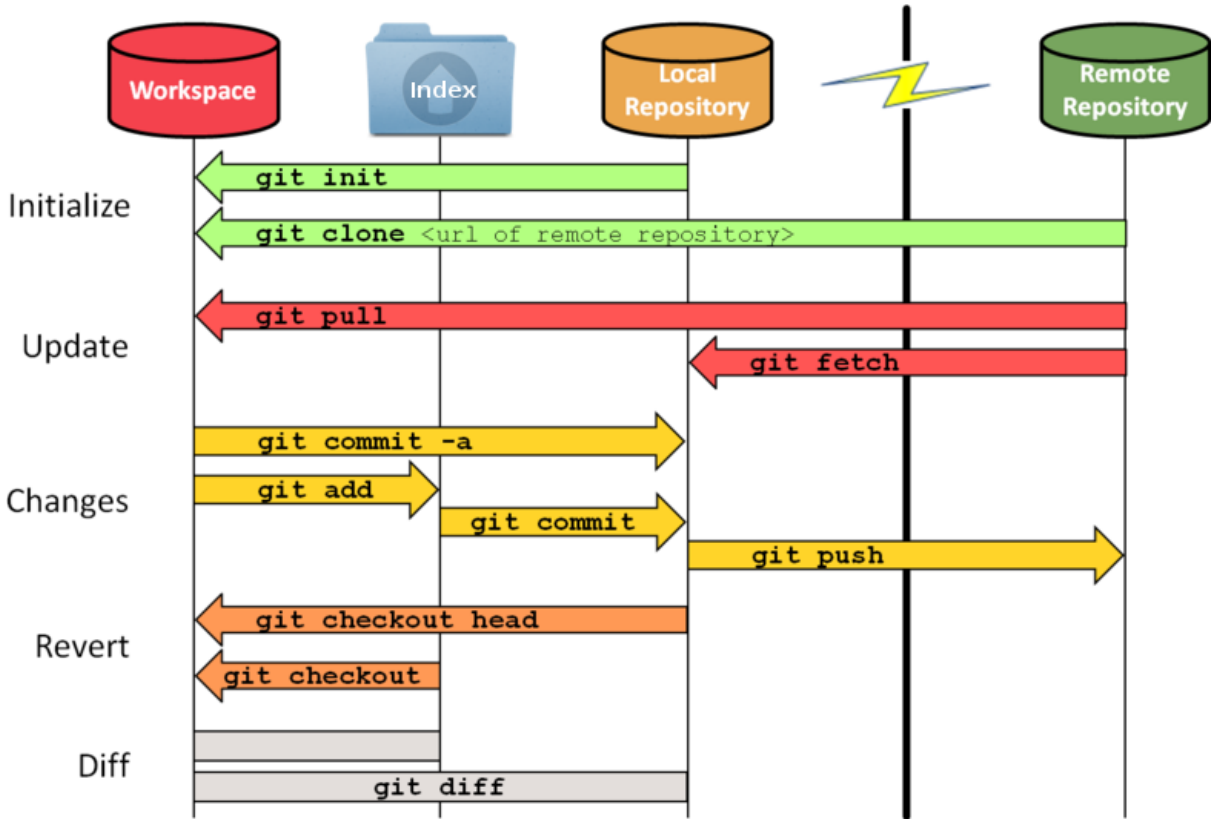
Contenu du script

Concepts fondamentaux de Git

Comme pour beaucoup d'outils informatiques, l'anglais est la langue de prédilection, et certains termes ne sont pas traduits car utilisés tels quels au quotidien, y compris par les développeurs francophones.

Pour un dossier de travail donné, Git manipule différents « espaces virtuels » :

- ***Workspace*** : espace stockant les modifications en cours, **qui ne sont pas (encore) prises en compte** par Git.
- ***Index (ou stage)*** : espace stockant les modifications en cours, **qui seront prises en compte par Git pour le prochain commit (mais pas encore commitées)**.
- ***Local repository*** : espace stockant les modifications **déjà prises en compte par Git**.
- ***Remote repository*** : désigne le dépôt distant (*remote*, sur GitHub par exemple), dépôt auquel est relié votre dépôt local. **Les commits du *local repository* doivent y être pushés pour mettre le dépôt distant à jour** et collaborer avec d'autres personnes.



Mémo de commandes Git

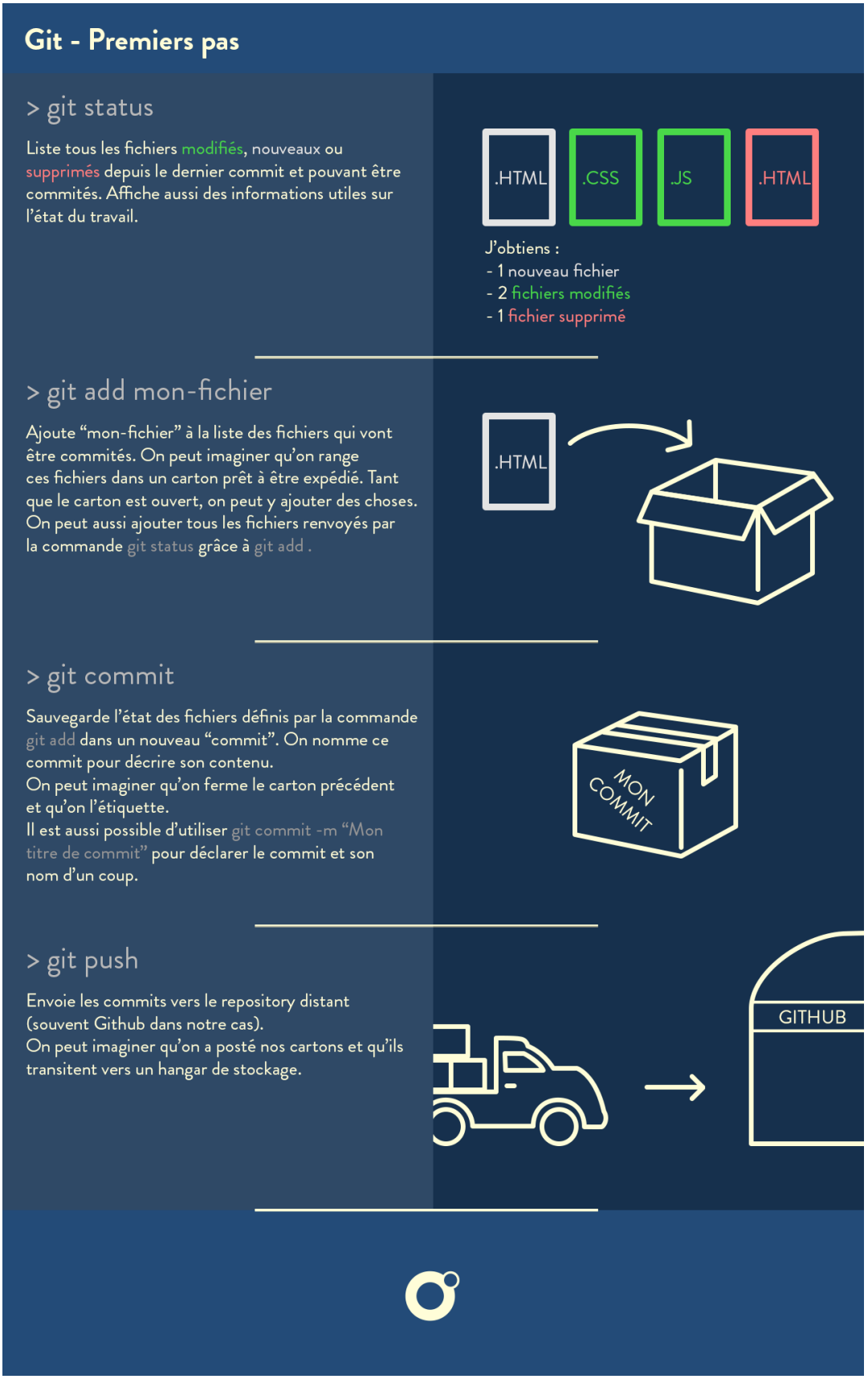
Obtenir de l'aide

- git --help : renvoie la liste des commandes disponibles
- git [commande] -h : idem pour une commande précise

Cloner ou initier un repo

- git clone {url} [nom-local] : récupère un repo distant (*remote* sur GitHub par exemple) en local, dans un dossier créé à la volée qu'il est possible de renommer (par défaut : nom du repo sur le *remote*)
- git init : crée un nouveau projet Git local à partir d'un dossier courant. Si on veut ensuite le partager sur GitHub, il faudra alors paramétrer au moins un *remote*

Mémo pour les premiers commits



Vérifier l'état courant du repo local

- git status : récapitule l'état local (workspace et index) des fichiers du projet géré avec Git
    - En rouge : modifié mais non pris en compte (= en workspace)
    - En vert : modifié et pris en compte (= ajouté à l'index)

Consulter l'historique du repo local

- git log : voir l'historique de tous les commits de la branche actuelle
- git log --oneline : idem en version synthétique
- git log --graph : idem en version branches

Consulter ou se déplacer dans les commits

- git show hashDuCommit : montre les modifs dans un commit précis – hashDuCommit est les 7 premiers caractères de son identifiant (ex : 1fd88ar)
- git checkout hashDuCommit : permet de consulter la version hashDuCommit du projet /!\ si on fait des modifs en version antérieure, elles seront perdues !
- git checkout master : revient sur la dernière version en date du projet sur cette branche !
  master = bout de la branche (le + récent) /!\ avant un pull, l'origin/master n'est pas le même que notre master local si d'autres contributeurs ont ajouté des commits depuis notre dernier pull !
- git checkout HEAD nomDuFichier : annule localement tous les changements sur le fichier depuis le dernier commit
  HEAD = pointeur dans la branche (commit qu'on est en train de consulter : identique ou antérieur à master)
- Annuler un add + commit *non pushé* : git reset HEAD^
- Supprimer un commit  **cela va supprimer vos modifications en local également**:
    - Revenir en local sur le dernier commit correcte : git reset --hard hashDuCommit
    - Forcer cette version comme étant la dernière : git push -f

Branches

- git branch nomDeLaBranche : crée une nouvelle branche (nouvelle version du projet) identique à la version HEAD – permet de tester des développements (réparer un bug, expérimenter une nouvelle fonctionnalité…) et commiter sans affecter la branche principale
- git checkout nomDeLaBranche : bascule sur la branche nomDeLaBranche
- git checkout -b nomDeLaBranche : permet de réaliser les deux opérations du dessus (création de la branche & migration sur celle-ci)
- git merge nomDeLaBranche : rapatrie les commits de la branche ciblée sur ma branche actuelle ==> nécessite de résoudre les conflits !

Gestion des fichiers

- git add [filename] : ajoute les modifications faites dans le fichier à l'index
- git add ., *alias, technique du bourrin* : ajoute les modifs de tous les fichiers dans le dossier
  /!\ à n'utiliser si on est sûr de n'avoir fait QUE des modifs liées au commit en cours !
- git reset HEAD : annule les add déjà faits
- git commit [-m "message de commit"] : enregistre les modifs indexées dans le commit
- git push : envoie tous les commits locaux sur le repo
- git pull : récupère en local un projet depuis le repo, pour un projet déjà en cours (au contraire de clone qu'on utiliser uniquement pour un nouveau projet)
- git restore . : permet d'annuler toutes les modifications effectuées sur les fichiers locaux
- git rm --cached <file(s)> : **supprimer des fichiers suivis du repo** qui auraient été ajoutés par erreur OU **ajoutés dans le .gitignore par la suite** (ne supprime pas le fichier lui-même en local, bien entendu, uniquement dans le dépôt).

En cas de suppression de votre dossier .git (cloné depuis github)

```
git init
git remote add origin ssh@le/lien/vers/ton/repo.git
git add .
git commit -m "on envoiiiie"
git push origin --force master
```

En cas de dépôt local corrompu

Si vous avez des messages du type : error: object file .git/objects/31/65329bb680e30595f242b7c4d8406ca63eeab0 is empty ou fatal: loose object 3165329bb680e30595f242b7c4d8406ca63eeab0 (stored in .git/objects/31/65329bb680e30595f242b7c4d8406ca63eeab0) is corrupt. C'est que votre dépôt local est corrompu. Voici la manoeuvre pour réparer :

```
find .git/objects/ -type f -empty | xargs rm
git fetch -p
git fsck --full
```

Si cela ne suffit pas, faites :

```
git reflog
```

Si vous avez ce message **fatal: bad object HEAD**, faites :

```
tail -n 2 .git/logs/refs/heads/master
```

Cela vous permet de récupérer l'id du dernier commit. Vous pouvez placer le HEAD dessus maintenant :

```
git update-ref HEAD [à remplacer par l'id du dernier commit effectué]
```

En cas de bêtise…

Parce que oui, ça arrive de se rendre compte qu'on doit revoir notre code, après l'avoir scellé dans un coffre lui-même enfermé dans une pièce secrète, elle même gardée par des mercenaires armés jusqu'aux dents, eux-mêmes … Bon, tu as compris l'idée, on ne peut plus revenir sur nos commits une fois créés.

Mais en fait, si, on peut. Il suffit juste de connaître les commandes.

Voici donc 2 ressources qui expliquent bien ces commandes, et dans quel cas les utiliser. A toi d'utiliser celle que tu préfères

- https://ohshitgit.com/fr (version originale)
- https://dangitgit.com/fr (version + « élégante »)

Fichiers spéciaux

- .gitignore : permet de lister des fichiers qui doivent être ignorés lors du add
- .keep : est un fichier qui peut être placé à la racine d'un répertoire vide afin que git prenne ce dossier en compte même s'il ne contient pas de fichier