INTRODUCTION À GIT

Table des matières

A) Introduction

GIT est un **système de révision de code source**, inventé par *Linus Torwalds*, le créateur originel du noyau *Linux*, noyau qu'il convient de bien distinguer du système d'exploitation *GNU/Linux*, lui-même constitué de logiciels libres GNU, sous licence GPL, contributions de *Richard Matthew Stallman*.

Il répond au besoin de **travailler à plusieurs développeurs sur un même projet**, au moyen de *branches*, et de *dépôts* (= espaces de stockage), lesquels peuvent être locaux (dans l'entreprise) ou distants (sur internet), tout en gardant l'historique des modifications réalisées sur les fichiers.

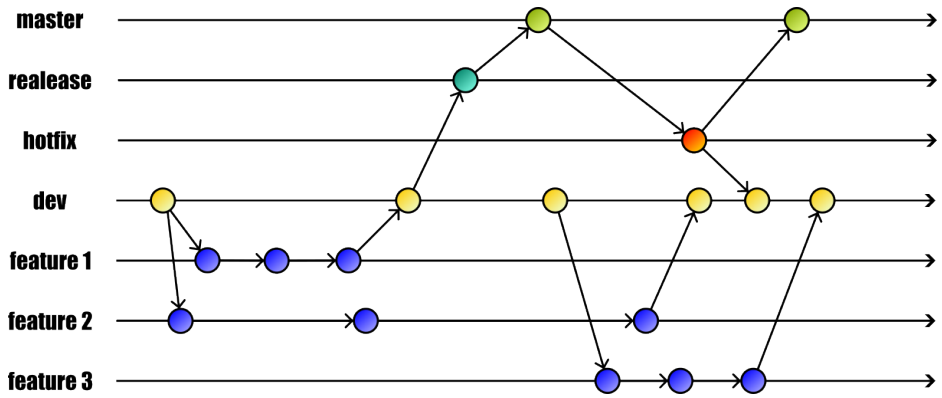
Mais GIT ne se contente pas de mémoriser qui à fait quoi et quand : il permet réellement de revenir en arrière dans son code, de réintégrer des changements passés, de fusionner des branches, etc en laissant aux développeurs la gestion des conflits, etc.

L'outil est pensé pour être très souple à l'usage, et n'impose aucun schéma définitif aux développeurs, juste des lignes directrices à suivre.

Il n'y donc aucune Intelligence Artificielle dans GIT : c'est aux développeurs de se mettre d'accord sur les branches à créer pour leur projet, et de s'organiser sur la manière de les exploiter.

Dans l'exemple suivant : on a

- ➔ une branche **master** qui existe par défaut, et qui contient les versions publiques officielles du projet
- ➔ une branche **release** dédiée en interne aux développeurs pour parfaire la version officielle avant sa sortie
- ➔ une branche **dev** de développement principale, sur laquelle il ne faut JAMAIS travaillé directement
- ➔ des branches dédiées aux fonctionnalités (**feature**) qui peuvent être courtes ou longues



Une chose très déstabilisante pour quelqu'un qui découvre GIT la première est que **l'outil rend les dossiers et fichiers du projet dynamiques**.

Ainsi si à un moment donné vous revenez en arrière dans votre code source, vos fichiers seront automatiquement réécrits à la date retenue dans le dossier de travail !

Rassurez-vous : vous ne perdrez pas vos dernières modifications : il suffira de revenir sur la dernière révision (COMMIT), pour retrouver tous vos fichiers actuels à leur emplacement d'origine.

Comprenez que sous GIT, **votre dossier de travail devient virtuel**.

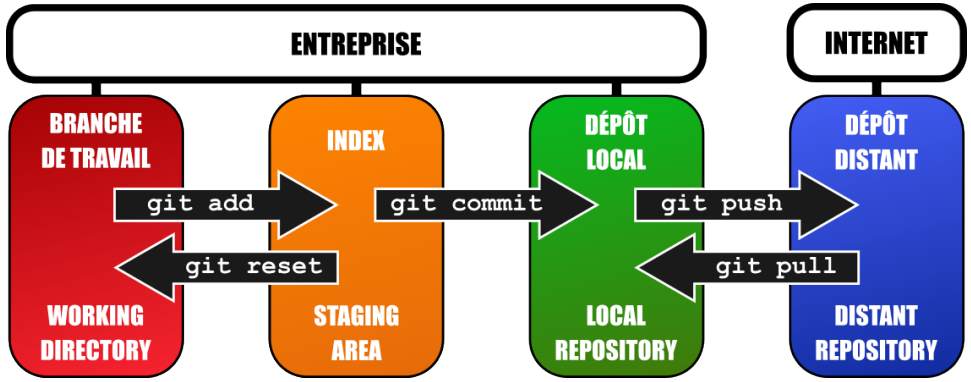
Et attention également à **ne pas confondre GIT avec un système de sauvegarde automatique**. Il peut très bien arriver que vous fassiez de grosses erreurs de manipulation en cours de travail, même pour un pro de l'outil !

Ainsi GIT ne vous empêchera pas de le suicider courageusement...

Donc si vous voulez être tranquille, n'hésitez pas à conserver des sauvegardes incrémentales classiques !

Sous GNU/Linux, avec 20 lignes de scripts, la commande **rsync** vous permet ainsi d'automatiser, via **crontab** ou les **timer systemd** une sauvegarde automatique particulièrement efficace et optimisée, locale ou distante.

B) Logique de base du dossier de travail



Grossièrement, on peut dire qu'il faut raisonner sous GIT avec 5 éléments :

➔ la **branche de travail courante**, avec vos fichiers et dossiers « virtuels » en cours de modification.

➔ **l'index GIT**, qui est une espèce de zone tampon, reprenant tous les fichiers/dossiers modifiés dans la branche courante, en attente d'une nouvelle révision (ou COMMIT).

➔ **le dépôt local** qui contient la copie locale du projet sous forme de branches et de révisions. Ce dépôt peut être désynchronisé du dépôt distant si on travaille sur plusieurs sites physiques, et peut donc contenir des branches qui n'ont pas encore été mises à jour.

➔ **le dépôt distant** qui contient normalement l'ensemble des branches à jour pour tous les participants, à l'exception des modifications apportées de votre côté, qui soit n'ont pas été encore révisées, soit n'ont pas encore été envoyées sur le dépôt distant.

➔ le **pointeur HEAD en local**, qui pointe toujours sur la branche de travail active (**master** au départ du projet). On y reviendra plus loin...

Tout l'art du développeur est donc de comprendre où en sont ces 5 éléments de base, à chaque commande tapée.

Et si l'outil est puissant dans la pratique, les subtilités sont ici très nombreuses, car elles couvrent tous les cas de figure dans une équipe de développement...

C) Au secours !

```
git status
git log --oneline
```

D) Installation des outils sous Debian 12

Nous allons ici utiliser le paquet ***git*** de GNU/Linux, installé avec les commandes habituelles :

```
apt update && apt -y install git npm gitg
```

mais également un outil plus visuel nommé ***ungit***, sous ***node.js***, qui permet de bien comprendre l'architecture de l'outil.

```
npm install -g ungit
```

E) Au début était le dépôt GIT

Un « dépôt » est un lieu de stockage (`REPOSITORY`) des fichiers du projet.

Sous GIT, tout contributeur possède par défaut son propre **dépôt local**, hébergé sur sa machine de travail, soit qu'il l'ait créé de toute pièce via `git init` dans son dossier de travail, soit qu'il l'ait téléchargé via `git clone` ou `git pull` (on verra ces commandes plus loin).

Le dépôt local peut contenir tout ou partie des branches du projet, avec un historique complet ou partiel.

L'idée de base est que le contributeur va travailler en local sur sa machine, sur une ou plusieurs branches du projet, voire en créer de nouvelles, puis va envoyer ses modifications vers un **dépôt distant**.

ce dépôt distant n'est pas forcément un site hébergé sur le cloud (chez **gitlab**, **github** ou autres). Il peut tout à fait rester interne à l'entreprise, donc privé. Mais la logique reste la même : le dépôt distant sert de stockage, et on ne peut pas travailler directement dessus. On ne peut qu'y télécharger/téléverser des fichiers.

À savoir : à la racine du dossier de travail, le dossier caché ***.git*** contiendra l'ensemble des branches locales du projet. Il ne faut donc surtout pas modifier, effacer ou déplacer ce dossier !

1) Cloner ou actualiser, il faut choisir !

Attention à ne pas confondre les options **pull** et **clone** !

➔ `git clone` télécharge l'ensemble des fichiers d'un projet distant en local en écrasant les fichiers existants ! On l'utilise donc une seule fois, pour ramener un projet existant, soit pour y contribuer, soit juste pour l'utiliser rapidement (par exemple pour compiler le pilote d'un dongle WiFi/Bluetooth sous GNU/Linux).

➔ `git pull` est un alias de `git fetch` + `git merge`. La commande télécharge le différentiel entre dépôt distant et dépôt local. C'est cette commande qui sert à travailler avec d'autres contributeurs, et permet de fusionner régulièrement les changements opérés par ces autres membres sur sa machine. Plus précisément, `git fetch` récupère l'index des fichiers/dossiers SANS modifier le dépôt local. `git merge` fusionne les modifications distantes dans le dépôt local, avec l'aide de l'usager en cas de conflits de code.

➔ Exemples :

```
git clone
ssh://bob@monprojet.com/chemin/vers/monprojet.git

git pull
https://gitlab.com/a1234/nom_du_projet.git
```

N.B. : on utilisera souvent `git pull` pour ramener une branche précise du dépôt. exemple :

```
git pull ALIAS_DEPÔT NOM_BRANCHE
```

F) Sont ensuite apparues les « branches »

Il faut voir une branche comme l'historique du projet, où apparaît chaque révision.

Par défaut, GIT nommera la branche principale **master**, mais on peut modifier ce nom à la création du projet (ici en **main**) :

➔ Renommer la branche principale à la création du dépôt :

```
git init --initial-branch=main
```

De là, deux cas de figure :

➔ Soit vous travaillez seul sur le projet, et en général, vous n'aurez pas besoin de créer d'autres branches.

➔ Soit vous travaillez à plusieurs et en même temps sur le projet. Et là, il faut éviter de « casser » le code source de la branche principale, à coup de révisions des uns ou des autres ! Ainsi sur un gros projet, on pourra créer des branches **hotfix**, **support**, **revision**, **dev**, et plusieurs **feature/nom_fonctionnalite**, ...

On peut bien entendu aussi créer des branches par développeur ou groupe de développeurs. Tout est libre ! À vous de trouver la meilleure organisation pour votre projet !

Chaque nouvelle branche utilisera le code source tel qu'il était à l'instant de sa création. Une fois que les contributeurs ont mis au point et finalisé leurs changements dans leurs branches respectives, ils viendront fusionner (**merge**) leurs branches à une branche parente, jusqu'à la branche principale **master**.

Attention : lors de la fusion de branches, quand GIT détecte que deux fichiers diffèrent , il avertit simplement les développeurs du conflit en leur indiquant les deux versions. C'est donc à ces derniers que revient la tâche ingrate de faire le tri, et d'assurer la fusion finale !

➔ Afficher les branches locales :

```
git branch
```

➔ Afficher les branches distantes :

```
git branch -r
```

➔ Afficher toutes les branches :

```
git branch -a
```

➔ Créer une branche locale :

```
git branch NOM_BRANCHE
```

➔ Supprimer une branche locale :

```
git branch -d NOM_BRANCHE
```

Attention : supprimer une branche revient à de fait supprimer son historique…

➔ Changer de branche de travail :

```
git switch/checkout NOM_BRANCHE
```

➔ Revenir à la branche principale :

```
git switch/checkout master
```

Il faut ici bien comprendre que `git switch/checkout` déplace simplement le pointeur HEAD vers la nouvelle branche de travail, sans modifier le pointeur actif de la branche de départ (sous-entendu : si vous revenez à la branche d'origine, vous vous

retrouverez au même endroit qu'au moment où vous l'avez quittée).

1) Fusion d'une branche

On admettra ici qu'on a une banche **main** et une branche **dev** pour le développement.

Il y a deux modes de fusion possibles.

➔ Soit la branche **dev** est en avance sur la branche **master**, laquelle n'a pas bougée depuis la création de la branche **dev**. Dans ce cas, la fusion se fera en mode **fast forward** (avance rapide), qui consiste simplement à « coller » les révisions de **dev** dans **master**.

➔ Dans tous les autres cas, ça veut dire qu'il y a eu des changements dans **master** provenant d'autres origines, et que notre version de **master** est donc désynchronisée. Dans ce cas, la fusion de notre branche **dev** dans master sera forcément manuelle, et vous aurez sûrement quelques conflits de code à gérer...

La fusion se fait toujours en se plaçant sur la branche de destination, qu'il faut donc préalablement sélectionner avec :

```
git switch/checkout NOM_BRANCHE_DESTINATION
```

soit ici :

```
git switch/checkout master
```

Puis on lance la fusion, ici de **dev** dans **master** :

```
git merge NOM_BRANCHE_SOURCE
```

soit ici :

```
git merge dev
```

G) Configuration de GIT

Elle est stockée dans le ***~/.gitconfig*** de votre utilisateur.

➔ Quelques réglages de base :

```
git config --global user.name "Votre prénom/nom"
git config --global user.email "Votre email"
git config --global core.editor nano ou vim
```

➔ Lister la configuration globale

```
git config --list
```

➔ Voir un paramètre en particulier

```
git config user.name
```

On peut aussi créer des alias dans GIT avec :

```
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.ad add
git config --global alias.st status
git config --global alias.br branch
```

```
git config --global alias.unstage 'reset HEAD --'
…
```

lesquels seront enregistrés dans ***~/.gitconfig***. Attention donc à ne pas effacer ce fichier caché en cours de travail !

Cela étant, dans la pratique, il est souvent plus facile de se créer de vrais alias shell dans un ***~/.bash_profile*** et de rajouter la ligne :

```
. ~/.bash_aliases
```

à la fin de son ***~/.bashrc***., ce qui nous donne par exemple :

```
alias gsw='git switch'
alias gc='git commit'
alias ga='git add'
alias gs='git status'
alias gb='git branch'
alias gr='git reset HEAD --'
…
```

À vous de trouver votre style et vos alias !

H) Création d'un dépôt

1) Depuis zéro

En local, on peut créer deux types de dépôts :

➔ un dépôt **normal**, qui va simplement ajouter un dossier caché ***.git*** dans votre dossier de travail courant. Ce dernier peut déjà contenir les fichiers/dossiers de votre projet à l'initialisation. Il suffit alors de se positionner à la racine de votre dossier de travail, et de faire un simple :

```
git init
```

➔ un dépôt « **bare »** (nu), qui va uniquement stocker la structure du projet GIT seule. Un dépôt **bare** ne permet donc pas de travailler directement sur les fichiers/dossiers : **c'est un dépôt local** destiné uniquement à accueillir et partager les contributions de tous les développeurs du projet, exactement comme un dépôt distant hébergé sous GitLab ou autre. **Il faut alors se positionner obligatoirement dans un dossier VIDE**, et faire la commande :

```
git init --bare
```

Notez bien ici un des avantages indéniables de GIT : l'outil ne vous force en rien à utiliser des dépôts extérieurs. Vous pouvez faire tous vos développements en interne, avec exactement les mêmes fonctionnalités !

2) Depuis un dépôt distant

Parmi les nombreuses autres possibilités offertes par GIT, et celles déjà vues avec **push**, on retiendra encore :

```
git clone REPOSITORY.git NOM_DOSSIER_LOCAL
```

pour cloner un dépôt en mode normal, et

```
git clone --bare REPOSITORY.git NOM_DOSSIER_LOCAL
```

pour cloner un dépôt en mode **bare**, le `push` sur un dépôt **bare** provoquant la mise à jour du dépôt original.

Attention : comme dit précédemment, la commande `clone` n'est à faire qu'au tout début du projet, pour initialiser le dépôt. Pour ramener ensuite les modifications d'autres développeurs sur un dépôt **normal**, c'est forcément `pull` qu'il faudra utiliser !

I) Gestion du cache local

1) Ajouter des fichiers avant soumission

On prendra ici appuie sur un dépôt normal, en admettant que le développeur a déjà des fichiers/dossiers à réviser.

➔ Transférer des fichiers/dossiers dans l'**index** :

```
git add FICHIERS_ET_OU_DOSSIERS
```

➔ Transférer tous les changements depuis le dossier courant, SANS les fichiers/dossiers effacés :

```
git add .
```

➔ Faire la même chose AVEC les fichiers/dossiers effacés :

```
git add -A
```

ou

```
git add --all
```

2) Ignorer des fichiers/dossiers dans le dossier de travail

Il n'est pas rare quand le dossier de travail, on ne veuille pas transférer certains fichiers locaux.

Dans ce cas, il suffit de créer un fichier caché **.gitignore** à la base du dossier de travail, et d'y placer la liste des fichiers/dossiers que GIT doit ignorer dans la phase d'ajout au cache.

La syntaxe du fichier est très simple :

```
.gitignore
/autre_nom_fichier_ou dossier_a_ignorer
/img/*.webm
etc. (ne pas rajouter cette ligne, évidemment…;)
```

3) Enlever des fichiers du cache local avant soumission

Si on fournit à la commande `reset` un chemin vers un fichier/dossier, elle ne touche pas au pointeur **HEAD**, ni à la branche pointée. Elle se contente de travailler sur l'index ou le dossier de travail.

➔ Désindexer un fichier/dossier ajouté via `git add` !

```
git reset HEAD FICHIER_OU_DOSSIER
```

```
git reset -- FICHIER_OU_DOSSIER
```

Sans arguments, la commande utilise la branche pointée par **HEAD**, en mode `--mixed` (voir explications plus loin), autrement dit elle ne touche qu'à l'index et pas au dossier de travail.

➔ Rechercher le fichier/dossier d'une révision précédente, et en faire la nouvelle référence dans l'index :

```
git reset NUM_COMMIT -- FICHIER_OU_DOSSIER
```

là encore, sans toucher au dossier de travail, à moins d'utiliser l'option `--hard` (voir explications plus loin), qui elle va réellement remplacer en sus le fichier/dossier existant par son ancienne version, sans possibilité de retour en arrière !

J) Revenir à une ancienne version dans le dossier de travail

Comme on l'a vu précédemment, il suffit d'utiliser l'option `--hard` de la commande `reset` pour rechercher une ancienne version de fichiers/dossiers, en écrasant définitivement les fichiers/dossiers existants dans notre dossier de travail.

Soyez donc très prudents quand vous utilisez cette option !

K) Les révisions

1) Journal

➔ Afficher toutes les révisions :

```
git log
```

➔ N'afficher que les numéros des révisions :

```
git log --oneline
```

➔ Voir le journal concernant un fichier :

```
git log -p readme.md
```

➔ Voir le journal concernant les deux dernières modifications d'un fichier :

```
git log -p readme.md -n 2
```

2) Gestion globale

➔ Envoyer une nouvelle révision avec un commentaire :

```
git commit -m "COMMENTAIRE_DE_LA_RÉVISION"
```

➔ Zut : j'ai (encore) oublié de transférer un fichier/dossier dans la dernière révision...

```
git add FICHIER_OU_DOSSIER_ALZHEIMER
git commit --amend
```

➔ Supprimer une révision antérieure, tout en gardant les révisions qui ont suivi :

```
git revert NUM_REVISION
```

➔ Revenir réellement à une révision antérieure du projet, en

perdant toutes les modifications qui ont été faites ultérieurement (dangereux) :

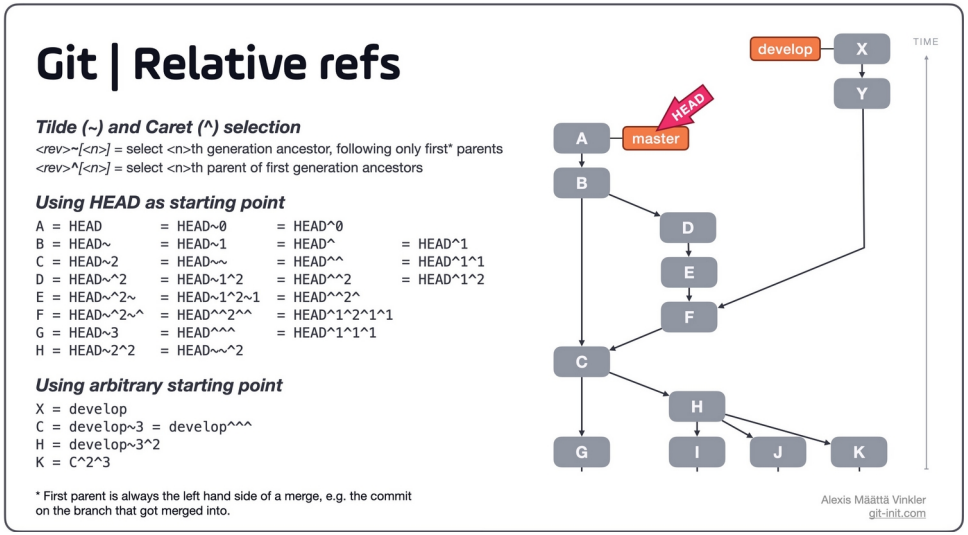
```
git reset NUM_REVISION --hard
```

3) Option reset sur une révision

Admettons implicitement que notre projet a déjà plusieurs révisions du code à son actif...

On voudra ici revenir à la révision précédente, et plutôt qu'un numéro de révision, on prendra ici le raccourci **HEAD~**.

Attention à la signification des ^ et des ~. Les ^ pointent vers les parents de la révision qui peuvent être multiples, les ~ visent la lignée directe de la révision !



Comparons les trois modes de fonction de `reset` en l'absence de chemin vers des fichiers/dossiers :

➔ Modification de **HEAD+branche** mais pas de l'index en cours ni des fichiers/dossiers de travail.

```
git reset --soft HEAD~
```

➔ Modification de **HEAD+branche** + **index**, mais pas des fichiers/dossiers de travail (comportement par défaut).

```
git reset --mixed HEAD~
```

➔ Modification de **HEAD+branche** + **index** + **FICHIERS/DOSSIERS** ! (sans possibilité de retour en arrière).

```
git reset --hard HEAD~
```

Pour mieux comprendre ce fonctionnement, le lien suivant en français vous permettra d'avoir un aperçu plus graphique des modifications apportées :

https://git-scm.com/book/fr/v2/Utilitaires-Git-Reset-démystifié

L) Retour vers le futur (checkout)

Nous avons vu que l'option `reset` de GIT permettait de désindexer des fichiers mis dans le cache par erreur, et d'écraser un fichier/dossier existant par une ancienne version.

L'option `checkout` est un peu différente : elle va surtout permettre au développeur de consulter une ancienne révision, en déplaçant le pointeur **HEAD**.

Pendant la consultation, les fichiers/dossiers existants, dans le dossier de travail, sont remplacés par leurs anciens homologues. Il ne faut donc pas paniquer devant ces changements temporaires.

En fin de consultation, le retour à la branche **master** restitue les fichiers originaux.

➔ Consultation d'une révision antérieure :

```
git checkout NUM_REVISION
```

➔ Retour à la branche main courante :

```
git checkout master
```

Attention : si en mode `checkout`, vous apportez des modifications à un fichier ou dossier, vous quittez la branche sur laquelle vous êtes (et non : « quitter » n'est pas « scier »...).

Vous vous retrouvez donc dans un mode « HEADLESS », comprenez une branche « virtuelle », et GIT ne saura pas quoi faire de vos changements, si vous ne lui indiquer pas clairement ce que vous voulez en faire...

Un `git status` vous indiquera alors les possibilités offertes.

M) Intégration avec GitLab

GitLab est aujourd'hui l'une des plateformes les plus employées, depuis le rachat de GitHub par Microsoft qui a vu la migration de nombreux développeurs libres, très déçus de s'être retrouvés du jour au lendemain, et sans avertissement aucun, chez le concurrent direct...

Pour un développeur avec de petits projets, il n'y a pas de frais d'hébergement. La plateforme propose comme sa concurrente des outils collaboratifs payants plus évolués pour les groupes de travail.

Vous pouvez donc ouvrir un compte sur GitLab, qui vous permettra de stocker vos projets publics – et surtout de vous faire la main sous GIT !

Pour éviter de devoir, à chaque envoi de révision, ressaisir votre nom d'utilisateur GitLab et son mot de passe associé, il vous faudra juste créer une paire de clés SSH (si vous n'en avez pas déjà), et transférer votre clé publique sur GitLab.

Nous allons voir juste après comment et surtout où sont stockés ces identifiants/

1) Identifiants d'accès

Deux possibilités !

➔ soit vous travaillez en équipe dans plusieurs dépôts distincts, exigeant chacun des identifiants différents pour envoyer vos révisions. Auquel cas les identifiants devront être stockés dans chaque dossier de travail respectif. Vous choisirez donc logiquement l'option de configuration :

```
git config credentials.helper cache
```

➔ soit vous travaillez toujours dans le même dépôt, sur un ou plusieurs projets, et là vous choisirez l'option de configuration :

```
git config --global credentials.helper cache
```

Dans les deux cas, au premier `push` venu, GIT vous demandera de saisir une fois vos identifiants, puis il les utilisera automatiquement par la suite.

P.S. : pour une configuration globale, les identifiants seront stockés directement dans ~/.git-credentials sous le format :

```
https://<USERNAME>:<PASSWORD>@gitlab.com
```

2) Fusionner un dossier de travail local avec un dossier GitLab fraîchement créé

Vus avez créé un projet sous GitLab, en lui donnant un nom, et vous constatez que par défaut, le dossier du nouveau projet contient deux fichiers, un fichier au format YAML, et un ***README.md*** qui affiche les renseignements généraux du projet.

Vous souhaiteriez maintenant convertir un dossier de travail sur votre machine, qui contient déjà les fichiers et dossiers du projet, pour en faire un dépôt GIT local à part entière, et le relier à votre hébergement distant :

```
git init --initial-branch=main
git remote add origin https://gitlab.com/a1234/nom_du_projet.git
git pull origin main --allow-unrelated-histories --no-rebase
```

3) Envoyer une révision sur un dépôt (après un commit)

```
git push -u origin main
```