

MEMO GitHub

19/03/2023 - 20/03/2023 - Création du dépôt sur Github

- Maj : 11/01/2026 -

<https://github.com/username/projectName/>

```
$ git config --global user.email      # Affiche le mail utilisé sur le compte
$ git config --global user.name      # idem pour le nom
```

```
$ git config --global user.name "username"      # Définit l'utilisateur
```

```
$ git config --global user.email "a@b.fr"      # Définit son mail
```

```
$ git config --global user.password "12345"    # Définit son mot de passe
```

```
$ git clone https://<Token>@github.com/username/projectName.git > clone le repository
distant dans le répertoire local courant. <Token> est le code de sécurité fourni par
Github, en remplacement de la traditionnelle paire { user, password }
```

Créer un nouveau dépôt local :

```
$ cd leRepertoireLocal ; git init
```

Pour créer un jeton qui permet l'accès au dépôt, à la commande git clone... :

Menu Profil > Settings > <>Developer settings > Personnel access > tokens > Tokens (classic)

```
$ cd projectName
```

Créer un commit :

```
$ echo « Bonjour » > README.md
```

```
$ git status > pour constater que le fichier a bien été créé, pas encore ajouté au dépôt.
```

```
$ git add README.md > ajoute le fichier dans l'espace de staging (index)
```

```
$ git status
```

```
$ git commit -m « ajout fichier README.md » > pour créer un commit à partir de l'espace de staging
```

```
$ git push > pour uploader le commit sur le dépôt distant
```

Créer un dépôt local puis distant (github):

```
cd rep_git ; git init
git add .
git commit -m « depot projet »
git remote add NomProjet https://github.com/<login\_user>/NomProjet.git
```

Initialisation dépôt Git

```
cd rep_du_projet_git
git switch --create main
```

```
git remote set-url origin git@gitlab.com:<user_login>/mon_projet.git
# redéfinit l'URL du dépôt
```

```
git remote -v
# pour check
```

```
# verification du bon fonctionnement :
```

```
git add .
git commit -m "Your commit message"
git push origin main
```

Initialisation du dépôt git pour un repo perso (Gitlab) :

```
cd existing_repo
git init --initial-branch=main
git remote add origin git@gitlab.com:<user_login>/mon_projet.git
git branch -M main // si la branche pas déjà créée
git add .
git commit -m "depot init"
git push -uf origin main
```

```
git push --set-upstream origin main
```

ajouter un dépôt (pour avoir gitlab + github :

```
git remote -v
git remote add github git@github.com:<gitUser..29>/deezerclone29.git
```

Opérations intéressantes

```
git push origin branch1:branch2
```

"Pousse la branche locale branch1 vers la branche branch2 sur le dépôt distant origin."

branch1 doit être une branche **locale** (celle que qu'on veut pousser).

branch2 doit être le nom de la branche **sur le dépôt distant** (celle qui va recevoir les modifications).

```
git push origin ma-branche-locale:nouvelle-branche-sur-origin
```

Si branch2 n'existe pas encore sur origin, elle sera créée automatiquement.

Réaliser un clone local branch2 à partir de branch1 :

```
git checkout branch2
```

On se place sur branch2.

```
git reset --hard branch1
```

Réinitialise branch2 pour qu'elle pointe sur branch1

Autre possibilité :

Supprimer et recréer branch2 depuis branch1 :

```
git branch -D branch2
```

Recréer branch2 depuis branch1 :

```
git checkout branch1  
git branch branch2
```

branch2 sera désormais un **clone parfait** de branch1, avec le même historique et le même contenu.

[Forcer un push :]

```
git push --force github main (ou origin dev...)
```

```
git push -force-with-lease --force-if-includes origin main
```

- **Sécurité** : --force-with-lease vérifie d'abord si l'état de la branche distante a changé avant de procéder. Si des commits ont été ajoutés par un autre contributeur, le *push* échouera, t'évitant ainsi d'écraser le travail des autres.
- **Conservation de l'historique** : en évitant d'écraser des commits, tu preserves l'historique des contributions, ce qui facilite le suivi des changements et la collaboration.

Force-if-includes :

Cette option permet de ne forcer le *push* que si ta branche locale inclut tous les commits de la branche distante. En d'autres termes, elle garantit que tu ne perdes pas accidentellement des commits qui pourraient être présents sur le dépôt distant.

Pour **mettre à jour branch2 avec les changements de branch1 sans perdre son historique** et **sans gérer de conflits** (quand on est le seul à travailler dessus), voici les meilleures approches :

1. Fusionner branch1 dans branch2 (sans conflit)

Si branch1 et branch2 ont divergé, mais que l'on veut conserver l'historique de branch2 tout en y intégrant les changements de branch1 :

```
git checkout branch2
git merge branch1
```

- **Avantage** : L'historique de branch2 est conservé, et les changements de branch1 sont ajoutés.
- **Cas idéal** : Si branch2 n'a pas de commits uniques que tu veux garder, ou si les changements de branch1 sont simplement un avancement linéaire (pas de divergence complexe).

2. Rebase branch2 sur branch1 (historique linéaire)

Si on veut un historique **linéaire** et éviter un commit de fusion :

```
git checkout branch2
git rebase branch1
```

- **Avantage** : L'historique reste propre et linéaire, et branch2 intègre tous les changements de branch1.
- **Attention** : Si branch2 a des commits uniques, ils seront "rejoués" sur branch1. Si ces commits dépendent d'un état ancien de branch1, des conflits peuvent survenir (mais comme tu es seul, tu peux les résoudre facilement).

Méthode	Conserve l'historique ?	Historique linéaire ?	Risque de conflit ?
git merge branch1	Oui	Non	Faible
git rebase branch1	Oui	Oui	Moyen
git reset --soft	Non (mais garde les changements non commités)	Oui	Aucun

Fusionne branch1 en acceptant tous ses changements :

```
git merge -X theirs branch1
```

L'option -X theirs indique à Git : *"En cas de conflit, prends toujours la version de branch1 (le côté 'theirs')."*

Attention : Cela écrase **tous** les changements de branch2 en cas de conflit.

Si on a déjà lancé un git merge et que tu es en plein milieu des conflits :

Accepter tous les changements de branch1 :

Marquer les conflits comme résolus :

Finaliser la fusion :

```
git checkout --theirs .
git add .
git merge --continue
```

Travail en commun

Fetch :

Les branches distantes sont importées dans le dépôt local avec un nom de la forme dépôtDistant/nomBranche

L'action « fetch » récupère dans le dépôt local les versions disponibles sur un dépôt distant et met à jour ou crée les branches de suivi

Merge :

On fusionne dans la branche courante la branche distante de même nom

Commit :

On archive le résultat de la fusion

Push :

On transfère sur le dépôt distant l'historique de la branche courante

L'action « push » transfère la branche locale sur le serveur. Si une branche de même nom n'existe pas sur le serveur, elle est créée. Sinon, si un « fast-forward » peut être effectué, la branche sur le serveur est mise à jour. Sinon, le « push » est refusé

Pull

L'action « pull » fait un « fetch » puis un « merge » dans une branche locale b à partir de la branche de suivi associée (b/d par défaut)

Travailler à plusieurs avec un dépôt commun

Principe de base

- Dépôt « en ligne » accessible par tous, au moins occasionnellement
- Chacun « clone » le dépôt commun sur sa machine

Périodiquement :

- Mise à jour de son dépôt local à partir du dépôt commun
- Transfert de versions stables de son travail dans le dépôt commun

Principe « évolué »

- Faire un « fork » du dépôt commun et travailler avec ce dépôt personnel distant
- Faire des « pull request » vers le dépôt commun lorsqu'on a atteint une version stable sur son dépôt personnel distant

En résumé...

● On travaille toujours avec 2 dépôts

- Le dépôt local
- Un dépôt « distant » (remote) auquel on associe un nom

● Les 4 principales commandes git permettant d'interagir avec un dépôt distant sont :

- **git fetch** : on récupère en local le contenu du dépôt distant
- **git pull** : équivalent d'un `git fetch` suivi d'un `git merge`
- **git push** : on transfère l'état actuel de la branche courante (+historique) vers le dépôt distant
- **git clone** : on crée un dépôt local à partir d'un dépôt distant



Commandes GIT de Base

Git config

- L'une des commandes git les plus utilisées est **git config**. On l'utilise pour configurer les préférences de l'utilisateur : son mail, l'algorithme utilisé pour diff, le nom d'utilisateur et le format de fichier etc. Par exemple, la commande suivante peut être utilisée pour définir le mail d'un utilisateur:

```
git config --global user.email sam@google.com
```

Consultation :

```
git config --get user.email
```

```
git config -l
```

Git init

- Cette commande est utilisée pour créer un nouveau dépôt [GIT](#) :

```
git init
```

Git add

- La **commande git add** peut être utilisée pour ajouter des fichiers à l'index. Par exemple, la commande suivante ajoutera un fichier nommé temp.txt dans le répertoire local de l'index:

```
git add temp.txt
```

Clone git

- La **commande git clone** est utilisée pour la vérification des dépôts. Si le dépôt se trouve sur un serveur distant, utilisez:

```
git clone alex@93.188.160.58:/chemin/vers/dépôt
```

- Inversement, si une copie de travail d'un dépôt local doit être créée, utilisez:

```
git clone /chemin/vers/dépôt
```

Git commit

- La **commande git commit** permet de **valider les modifications apportées** au HEAD. Notez que tout commit ne se fera pas dans le dépôt distant.

```
git commit -m "Description du commit"
```

Git status

- La **commande git status** affiche la liste des fichiers modifiés ainsi que les fichiers qui doivent encore être ajoutés ou validés. Usage:

```
git status
```

Git push

- **Git push** est une autre commandes [GIT](#) de base. Un simple push envoie les modifications locales apportées à la branche principale associée :

```
git push origin master
```

Git checkout

- La **commande git checkout** peut être utilisée pour créer des branches ou pour basculer entre elles. Par exemple nous allons créer une branche:

```
command git checkout -b <nom-branche>
```

- Pour passer simplement d'une branche à une autre, utilisez:

```
git checkout <nom-branche>
```

Git remote

- Cette commande **remote** permet à un utilisateur de se connecter à un dépôt distant. La commande suivante répertorie les dépôts distants actuellement configurés:

```
git remote -v
```

- Cette commande permet à l'utilisateur de connecter le dépôt local à un serveur distant:

```
git remote add origin <93.188.160.58>
```

Branche git

- La **commande git branch** peut être utilisée pour répertorier, créer ou supprimer des branches. Pour répertorier toutes les branches présentes dans le dépôt, utilisez:

```
git branch
```

- Pour supprimer une branche:

```
git branch -d <nom-branche>
```

Git pull

- Pour fusionner toutes les modifications présentes sur le dépôt distant dans le répertoire de travail local, la commande pull est utilisée. Usage:

```
git pull
```

Git merge

- La **commande git merge** est utilisée pour fusionner une branche dans la branche active. Usage:

```
git merge <nom-branche>
```

Git diff

- La **commande git diff** permet de lister les conflits. Pour visualiser les conflits d'un fichier, utilisez
`git diff --base <nom-fichier>`
- La commande suivante est utilisée pour afficher les conflits entre les branches à fusionner avant de les fusionner:
`git diff <branche-source> <branche-cible>`
- Pour simplement énumérer tous les conflits actuels, utilisez:
`git diff`

Git tag

- Le marquage est utilisé pour marquer des commits spécifiques avec des poignées simples. Un exemple peut être:
`git tag 1.1.0 <insert-commitID-here>`

Git log

- L' **exécution de** cette commande génère le log d'une branche. Un exemple de sortie :

```
commit 15f4b6c44b3c8344caasdac9e4be13246e21sadw
Author: Alex Hunter <alexh@gmail.com>
Date: Mon Oct 1 12:56:29 2016 -0600
```

Git reset

- Pour réinitialiser l'index et le répertoire de travail à l'état du dernier commit, la **commande git reset** est utilisée :
`git reset --hard HEAD`

Git rm

- **Git rm** peut être utilisé pour supprimer des fichiers de l'index et du répertoire de travail. Usage:
`git rm nomfichier.txt`

Git stash

- L'une des moins connues, **git stash** aide à enregistrer les changements qui ne doivent pas être commit immédiatement. C'est un commit temporaire. Usage:
`git stash`

Git show

- Pour afficher des informations sur tout fichier git, utilisez la **commande git show** . Par exemple:
`git show`

Git fetch

- **Git fetch** permet à un utilisateur d'extraire tous les fichiers du dépôt distant qui ne sont pas actuellement dans le répertoire de travail local. Exemple d'utilisation:

```
git fetch origin
```

Git ls-tree

- Pour afficher un fichier arborescent avec le nom et le mode de chaque élément, et la valeur SHA-1 du blob, utilisez la **commande git ls-tree** . Par exemple:

```
git ls-tree HEAD
```

Git cat-file

- À l'aide de la valeur SHA-1, affichez le type d'un fichier à l'aide de la **commande git cat-file** . Par exemple:

```
git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Git grep

- **Git grep** permet à un **utilisateur de rechercher** dans les arbres de contenu des expressions et / ou des mots. Par exemple, pour *rechercher* `www.hostinger.com` dans tous les fichiers, utilisez:

```
git grep "www.hostinger.com"
```

Gitk

- **Gitk** est l'interface graphique du dépôt local. Vous pouvez l'appeler en exécutant:

```
gitk
```

Git instaweb

- Avec la **commande git instaweb** , un serveur Web peut être exécuté par interface avec le dépôt local. Qui redirige directement vers un serveur web. Par exemple:

```
git instaweb -httpd=webrick
```

Git gc

- Pour optimiser le dépôt en supprimant les fichiers inutiles et les optimiser, utilisez:

```
git gc
```

git archive

- La **commande git archive** permet à un utilisateur de créer un fichier zip ou tar contenant les composants d'un arbre du dépôt. Par exemple:

```
git archive --format=tar master
```

Git prune

- Via la **commande git prune** , les fichiers qui n'ont pas de pointeurs entrants seront supprimés. Usage:

```
git prune
```

Git fsck

- Pour effectuer une vérification d'intégrité du système de fichiers git, utilisez la commande **git fsck** . Tous les fichiers corrompus seront identifiés:

```
git fsck
```

Git rebase

- La **commande git rebase** est utilisée pour la réapplication des commits sur une autre branche. Par exemple:

```
git rebase master
```

Migrer un repo (perso) vers un autre (de groupe)

1. Cloner le dépôt source

Depuis la machine locale (ou un serveur intermédiaire), il faut cloner son dépôt (privé dans l'exemple) avec toutes les branches et tags :

```
git clone --mirror git@gitlab.com:ton-utilisateur/ancien-repo.git
```

--mirror est important : il clone **tout** (branches, tags, refs, hooks, etc.).

2. Créer le dépôt de destination

- Dans GitLab, aller dans son **groupe / sous-groupe** → New project.
- Donner un nom au projet (par ex. nouveau-repo).
- Vérifier que les permissions correspondent à ce que l'on veut (visibilité, accès aux membres du groupe).

Cela donnera une nouvelle URL Git (quelque chose comme) :

```
git@gitlab.com:mon-groupe/mon-sous-groupe/nouveau-repo.git
```

3. Pousser tout le contenu dans le nouveau dépôt

Dans le dossier cloné :

```
cd ancien-repo.git  
git push --mirror git@gitlab.com:mon-groupe/mon-sous-groupe/nouveau-repo.git
```

Cela va pousser **toutes les branches, tags et l'historique** dans le repo du groupe.

4. Vérifications

- Vérifier sur GitLab que toutes les branches et tags sont bien présents.
- Tester un clone du nouveau dépôt pour s'assurer que tout est correct :

```
git clone git@gitlab.com:mon-groupe/mon-sous-groupe/nouveau-repo.git
```

Astuce : garder le dépôt original en lecture seule

Si on veut éviter que des contributions continuent sur l'ancien dépôt :

- Aller dans **Settings > General > Visibility, project features, permissions**.
- Mettre le repo en *archivé* → plus personne ne peut pousser dessus, mais l'historique reste accessible.

SOUS-MODULES

Asking IA :

sur un dépôt gitlab, j'ai créé un repository, avec un projet constitué de 2 dossiers principaux. Dans 1 de ces dossiers, il y avait déjà un répertoire `.git` après avoir initié le dépôt (dans le répertoire parent), puis `git add . / git commit / ...`. Je me rends compte que le second dossier est considéré comme un sous-module, qui apparaît sous cette forme dans Gitlab : `sousDossier @ d17a9e2d` au lieu d'être un dossier à part entière, contenant des fichiers et du code. Comment remédier à ce problème et remplacer le sous-module (sorte d'alias) par le "vrai" répertoire ?

GPT-4o mini

Pour résoudre le problème où un répertoire est traité comme un sous-module dans GitLab, vous pouvez suivre ces étapes pour le convertir en un véritable répertoire dans votre dépôt. Voici comment procéder :

Étapes pour retirer un sous-module et le convertir en répertoire

1. **Supprimer le sous-module** : Tout d'abord, vous devez supprimer le sous-module de votre dépôt Git. Exécutez les commandes suivantes dans le terminal :

```
git rm --cached chemin/vers/sousDossier
```

Cela supprimera le sous-module de l'index Git tout en conservant les fichiers localement.

- Le cas échéant -

- **Supprimer la référence au sous-module** : Ouvrez le fichier `.gitmodules` à la racine de votre dépôt et supprimez la section qui correspond au sous-module. Cela ressemble généralement à ceci :

```
[submodule "chemin/vers/sousDossier"]
  path = chemin/vers/sousDossier
  url = url_du_sous_module
```

Enregistrez et fermez le fichier.

- **Faire un git add des changements** : Une fois que vous avez supprimé la référence, ajoutez les changements à votre dépôt :

```
git add .gitmodules chemin/vers/sousDossier
```

- **Commitez les changements** : Ensuite, engagez les modifications que vous venez de faire :

```
git commit -m "Retrait du sous-module et conversion en dossier normal"
```

- **Vérifier la structure** : Assurez-vous que le contenu de `sousDossier` est bien présent sans référence au sous-module. Vous devriez maintenant pouvoir travailler avec ce dossier comme un répertoire normal.
- **Pousser les modifications sur GitLab** : Finalement, poussez vos modifications vers GitLab :

```
git push origin main
```

Remplacez `main` par le nom de votre branche si besoin.

Résumé des commandes

Action	Commande
Supprimer le sous-module	<code>git rm --cached chemin/vers/sousDossier</code>
Modifier .gitmodules	Éditez et supprimez la section correspondante
Ajouter les changements	<code>git add .gitmodules chemin/vers/sousDossier</code>
Committer les changements	<code>git commit -m "Retrait du sous-module et conversion en dossier normal"</code>
Pousser les modifications	<code>git push origin main</code>

Une fois ces étapes réalisées, le dossier devrait apparaître correctement dans votre dépôt GitLab sans être considéré comme un sous-module.



Procédures utiles

synchro locale par rapport à un dépôt distant

Il manque 1 ou plusieurs branches d'un projet sur un PC local (qui n'est pas à jour)

=> **Récupérer la branche : branche_x qui existe sur le répo GitLab ?**

Check de l'ensemble des branches en locales (affichage branches locales + remote) :

```
git branch -a
```

Récup des dernières infos : mise à jour, localement, des références distantes :

```
git fetch origin
```

Création de la branche locale souhaitée, à partir de la branche distante :

```
git checkout -b branche_x origin/branche_x
```

Push vers plusieurs dépôts distants - Permission denied

un projet git local possède 2 dépôts distants : 1 sur GitHub, l'autre sur GitLab :

je souhaite faire un push vers le dépôt github (j'ai normalement une paire de clés public / privées associée à ce compte), mais ça me retourne l'erreur :

git@github.com: Permission denied (publickey). fatal : Impossible de lire le dépôt distant. Veuillez vérifier que vous avez les droits d'accès et que le dépôt existe.

L'erreur que vous rencontrez indique que votre clé SSH n'est pas reconnue par GitHub. Voici quelques étapes pour résoudre ce problème :

Vérification de la Configuration SSH

1. Vérifiez si votre clé SSH est ajoutée à l'agent SSH

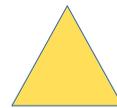
Ouvrez un terminal et entrez la commande suivante :

```
ssh-add -l
```

Si votre clé n'est pas répertoriée, vous pouvez l'ajouter avec :

```
ssh-add ~/.ssh/id_rsa
```

Assurez-vous de remplacer `~/.ssh/id_rsa` par le chemin correct de votre clé privée si elle n'est pas le nom par défaut.



2. Ajoutez votre clé SSH à GitHub

Si vous n'avez pas encore ajouté votre clé SSH à votre compte GitHub, faites ceci :

1. Copiez votre clé publique avec cette commande :

```
cat ~/.ssh/id_rsa.pub
```

Connectez-vous à votre compte GitHub.

- Allez dans **Settings > SSH and GPG keys**.
- Cliquez sur **New SSH key**, puis collez votre clé publique et donnez-lui un titre.

3. Vérifiez l'URL de votre dépôt GitHub

Assurez-vous que l'URL de votre dépôt distant est bien configurée pour utiliser SSH. Exécutez :

```
git remote -v
```

Vous devriez voir quelque chose comme :

```
origin git@github.com:<votre-utilisateur>/<votre-repo>.git (fetch)
origin git@github.com:<votre-utilisateur>/<votre-repo>.git (push)
```

Si ce n'est pas le cas, vous pouvez modifier l'URL avec :

```
git remote set-url origin git@github.com:<votre-utilisateur>/<votre-repo>.git
```

Essayer de se connecter à GitHub

Testez si la connexion SSH fonctionne en exécutant :

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

Vous devriez voir un message de bienvenue si la configuration est correcte.

4. Vérifiez les permissions de votre clé

Assurez-vous que les permissions de votre clé SSH sont correctes. Cela peut être fait avec :

```
chmod 600 ~/.ssh/id_rsa
chmod 644 ~/.ssh/id_rsa.pub
```

Conclusion

Après avoir effectué ces vérifications et modifications, essayez à nouveau de faire un git push vers votre dépôt GitHub. Si vous rencontrez toujours des problèmes, n'hésitez pas à me le faire savoir, et nous pouvons explorer d'autres solutions.

Supprimer un fichier qui ne devrait pas se trouver sur un dépôt git, pour toutes les branches :

```
git filter-repo --path fichier --invert-paths
```

Effet :

supprime le fichier > de toutes les branches > de tous les commits > tags inclus

=> Marche pas suivant la version de git

En fait : installer :

```
$ sudo apt install git-filter-repo
```

```
git filter-repo --use-base-name --path [FILENAME] --invert-paths
```

exemple :

```
git filter-repo --use-base-name --path googleAuth.asm --invert-paths --force
```

Autre solution : ajouter le/les fichiers au .gitignore, puis faire le nécessaire pour chaque branche et chaque outil de versioning (Github/gitlab) :

```
git switch dev
git rm --cached ms-n1/Dockerfile
git rm --cached docker-compose.yml
git add .
git commit -m 'maj ref docker'
git push origin dev
git push github dev
```

Suppression d'un fichier de l'historique du projet, définition d'un dépôt origin sur Gitlab et synchro de toutes les branches (local=>origin)

```
git filter-repo --use-base-name --path googleAuth.asm --invert-paths --force
git remote add origin git@gitlab.com:usergroup/gp_proj1/microservice/msnotif.git
git push origin --all
git push origin --tags
```