



# TP Git

## TP 1

 **Objectif** : créer un dépôt Git en local, le connecter à GitHub avec une clé SSH et publier son premier commit.

1. Préparer GitHub
  - Crée un compte sur [GitHub.com](https://github.com) si ce n'est pas déjà fait.
  - Sur GitHub, crée un **nouveau dépôt vide** (pas de README, pas de .gitignore) nommé tp1-projet.
2. Travailler en local
  - Sur ton PC, crée un dossier tp1-projet.
  - Initialise Git dans ce dossier (git init).
  - Crée un fichier journal.txt avec une phrase de ton choix.
  - Vérifie l'état du dépôt et fais un premier commit.
3. Configurer SSH
  - Génère une **clé SSH** sur ton PC.
  - Ajoute ta clé publique à ton compte GitHub .
  - Vérifie la connexion avec GitHub.
4. Connecter le remote
  - Relie ton dépôt local au dépôt GitHub en utilisant l'URL **SSH** du dépôt.
  - Vérifie que le remote est bien configuré.
5. Publier ton travail
  - Envoie ton commit initial sur GitHub
  - Vérifie que journal.txt est bien visible dans ton dépôt en ligne.

## TP 2 – Collaboration avec GitHub (pull vs fetch)

 **Objectif** : travailler en binôme pour comprendre clone, push, pull et découvrir la différence avec fetch.

1. Mise en place (Étudiant A)
  - Crée un dépôt GitHub nommé tp2-collab.
  - Ajoute un fichier membres.txt avec la phrase :  
**"Projet Git TP2 – par Étudiant A"**
  - Commit + push du fichier.
2. Étudiant B récupère le projet
  - Clone le dépôt GitHub (git clone).
  - Vérifie le contenu local (membres.txt).
3. Étudiant B contribue
  - Ajoute sa propre ligne dans membres.txt.
  - Commit + push.

4. Étudiant A met à jour
  - Avant d'utiliser pull, fais un git fetch origin.
  - Compare les différences entre **sa branche locale** et origin/main (utilise git log ou git diff).
  - Ensuite, utilise git pull origin main pour intégrer les changements de B.
  - Vérifie que la ligne de B apparaît bien.
5. Étudiant A ajoute sa ligne
  - Ajoute sa propre ligne dans membres.txt.
  - Commit + push.
6. Étudiant B met à jour
  - Étudiant B fait directement un git pull origin main.
  - Vérifie que membres.txt contient maintenant la ligne de A.

## TP 3 – Branches, Merge et Rebase

1. Crée une branche **feature1** et bascule dessus
2. Crée un fichier **exemple.txt**, écris une phrase dedans et fais un commit.
3. Vérifie que la branche existe aussi sur GitHub en la poussant
4. Crée une branche **branchtest**, puis supprime-la
5. Bascule sur feature1, modifie exemple.txt, commit.
6. Reviens sur main et merge feature1 :
7. Retourne sur feature1, fais une nouvelle modification, commit et push.
8. Mets à jour ta branche avec **rebase** :
9. Fais encore 3 commits rapides dans exemple.txt.
10. Lance un rebase interactif sur les 2 derniers commits :
  - git rebase -i HEAD~2




Un **fast-forward** est une fusion où Git avance simplement le pointeur de la branche, sans créer de commit de merge, car la branche principale n'a pas bougé depuis la création de la branche secondaire. 🏹 C'est utile car l'historique reste linéaire et propre, mais on peut l'éviter avec git merge --no-ff si on veut garder une trace explicite de la fusion.

## TP 4 – Jouer avec l'historique Git

1. Crée un petit projet
    - Dossier : tp4-historique
    - Fichier : notes.txt
    - Ajoute ces 3 lignes, une par commit :
      1. "Ligne A – première note"
      2. "Ligne B – deuxième note"
      3. "Ligne C – troisième note"
- 🏹 Vérifie avec git log --oneline que tu as 3 commits.


1. Fais un **reset**
  - Reviens au commit 1 avec :
  - Question : combien de commits vois-tu dans git log ?
  - Question : combien de lignes restent dans notes.txt ?
2. Fais un **revert**
  - Recommence avec les 3 commits initiaux (ou recommence le repo).
  - Annule uniquement le **commit 2**
  - Question : combien de commits vois-tu maintenant ?
  - Question : que contient le fichier notes.txt ?
3. Teste un **checkout <commit>**
  - Déplace-toi au commit 1
  - Question : que contient notes.txt maintenant ?
  - Question : que dit Git sur ton état (detached HEAD) ?
  - Bonus : si tu ajoutes une ligne "Ligne D – test" et commits, que se passe-t-il ?

## TP 5 - Mettre son travail de côté avec git stash

 **Objectif** : comprendre pourquoi Git bloque quand on change de branche avec des modifications non commit, et utiliser stash pour les mettre en pause.

1. Place-toi sur la branche feature1.
2. Crée un nouveau fichier teststash.txt.
3. Écris du texte dedans puis ajoute + commit.
4. Reviens sur main et fusionne feature1 avec main.
5. Reviens sur feature1.
6. Modifie teststash.txt (ajoute une nouvelle ligne) **sans commit**.
7. Essaie de passer sur main avec : git switch main
  - Git bloque (modifs risquent d'être écrasées).
8. Mets les changements de côté avec :
  - **git stash**
9. Vérifie la pile de stash :
  - **git stash list**
  - 🖱️ Tu vois ton travail mis de côté.
10. Reviens sur feature1.
11. Récupère le stash :
  - **git stash pop**
12. Commit les changements restaurés :
  - git add teststash.txt
  - git commit -m "Ajout modifs récupérées depuis stash"
13. Pousse ta branche sur GitHub :
  - git push origin feature1
14. Reviens sur main et merge feature1

## TP 6 – Résoudre un conflit

 **Objectif** : provoquer volontairement un conflit Git en travaillant à deux sur le même fichier, puis le résoudre ensemble.

## Étudiant 1

1. Crée un dépôt GitHub vide et ajoute un fichier histoire.txt avec une première ligne.
2. Crée une branche featureA.
3. Dans cette branche, ajoute une deuxième ligne dans histoire.txt.
4. Fusionne la branche featureA dans main et envoie le résultat sur GitHub.

## Étudiant 2

1. Clone le même dépôt GitHub.
2. Crée une branche featureB.
3. Dans cette branche, ajoute lui aussi une deuxième ligne différente dans histoire.txt.
4. Essaie de fusionner la branche featureB dans main.

## Conflit

1. Git détecte un conflit dans histoire.txt car les deux étudiants ont modifié la même ligne.
2. Ouvrez le fichier ensemble et discutez :
  - garder la version de l'un,
  - ou bien combiner les deux versions.
3. Supprimez les marqueurs de conflit ajoutés par Git.
4. Enregistrez la version corrigée, validez la résolution, et envoyez-la sur GitHub.

## Finalisation

1. Chaque étudiant met à jour sa copie locale pour vérifier que le fichier corrigé est bien identique.

# TP 8 – Préparer son profil GitHub comme portfolio

## Objectif

Avoir un profil GitHub propre, clair et professionnel, même avec peu ou pas de projets.

1. **Créer un profil propre**
  - Ajoutez une photo de profil simple (ou un avatar neutre).
  - Renseignez votre bio : *"Étudiant en informatique à Ynov, passionné par le développement ou/et la cybersécurité."*
  - Ajoutez vos coordonnées : mail étudiant ou LinkedIn.
2. **Créer un README de profil**
  - Créez un dépôt **au nom exact de votre pseudo GitHub**.
  - Dans le README.md, écrivez :
    - une présentation courte (nom, parcours, ce que vous aimez).
    - vos domaines d'intérêt (Go, Cyber, Data, DevOps...).
    - une phrase d'accroche fun ou geek (*"Toujours prêt à déboguer la vie !"*).
3. **Épingler un ou deux dépôts vides**
  - Exemple : go-projets ou cyber-labs.

- Ajoutez juste un README avec *“Projet à venir”*.
  - Cela montre que vous préparez déjà vos espaces de travail.
4. **Soigner vos futurs dépôts**
- Chaque projet que vous publierez (même petit TP) devra avoir :
    - un README clair,
    - une organisation simple (code + docs),
    - un message de commit propre.
5. **Bonus créatif**
- Ajoutez un badge *“Étudiant @ Ynov”* dans le README.

<https://www.youtube.com/watch?v=lfVVm5K-Xp0>

