TP guidé — Git : bases, dépôt local, synchronisation, .gitignore, branches & bonnes pratiques

> Durée indicative : 2–3 h · Niveau : débutant → intermédiaire · Modalité : individuel ou binôme

Objectifs pédagogiques

À la fin de ce TP, vous serez capables de :

- Configurer Git correctement sur votre poste.
- Créer un dépôt local, suivre des fichiers et enregistrer des versions (commits).
- Synchroniser avec un dépôt distant (push, pull, clone).
- Ignorer des fichiers avec `.gitignore`.
- Créer des branches, fusionner et résoudre un conflit simple.
- Appliquer des bonnes pratiques de versionnage.

Prérequis & installation

- **Git** installé (≥ 2.30). Vérifiez : `git --version`.
- Un compte **GitHub** ou **GitLab** (choisissez une plateforme et connectez-vous au navigateur).
- Un éditeur de code (ex. **VS Code**).
- Un terminal (PowerShell, bash, zsh…).

Conventions du TP

- Les commandes à exécuter sont précédées de `$` (ne pas taper le `$`).
- Les blocs *À faire* guident vos actions. Les blocs *Résultat attendu* précisent ce que vous devez observer/obtenir.
-

Partie 1 — Configuration initiale

1.1 Définir votre identité

Les métadonnées de commit (auteur, email) sont essentielles pour la traçabilité.

```
$ git config --global user.name "Prénom Nom"
$ git config --global user.email "prenom.nom@etu.univ.fr"
$ git config --global core.editor "code --wait"   # VS Code comme éditeur pour les messages
```

Explications

- `--global` enregistre la configuration dans `~/.gitconfig` (applicable à tous vos dépôts).
- `core.editor` ouvre l'éditeur pour les messages de commit/merge si vous n'utilisez pas `-m`.

À faire

1. Exécutez les trois commandes ci-dessus en remplaçant par vos informations.
2. Vérifiez la configuration :

$ git config --list

Résultat attendu
Vous voyez au moins `user.name`, `user.email`, `core.editor` correctement définis.

Partie 2 — Dépôt local : état des fichiers, staging, commit

2.1 Initialiser un dépôt

$ mkdir tp-git
$ cd tp-git
$ git init

Explications

- `git init` crée un sous-dossier caché `.git/` : c'est la base de données locale des versions.
- Votre répertoire de travail a 3 zones logiques :
  - **Working Directory** : vos fichiers sur disque.
  - **Staging Area (index)** : zone de préparation avant commit.
  - **Repository (.git)** : l'historique validé (commits).

À faire
Vérifiez l'état initial :

$ git status

Résultat attendu
Git indique : *No commits yet*, branche `main` (ou `master` selon version), aucun fichier suivi.

2.2 Créer un premier fichier et observer les états

$ echo "# Mon premier dépôt Git" > README.md
$ git status

Explications

- Le nouveau fichier `README.md` est **untracked** (non suivi).
- Pour l'inclure au prochain commit, on le **stage** avec `git add`.

$ git add README.md
$ git status

Résultat attendu
`README.md` apparaît en **staged** (vert) → prêt à être commité.

2.3 Créer un commit

$ git commit -m "Ajout du fichier README"

Explications

- Un **commit** = un instantané (snapshot) des fichiers **staged** + un message décrivant la modification.
- Le message doit être clair et à l'impératif court (ex. *Ajoute la page d'accueil*).

À faire
Consultez l'historique :

$ git log --oneline --decorate --graph --all

Résultat attendu
Vous voyez au moins 1 commit sur `main`. La graine du graphe (`*`) représente votre commit.

2.4 Modifier et comparer

$ echo "Une deuxième ligne" >> README.md
$ git diff # différences non stagées
$ git add README.md
$ git diff --staged # différences en staging vs dernier commit
$ git commit -m "Ajoute une deuxième ligne au README"

Explications

- `git diff` compare l'état de travail avec le dernier commit.

- `git diff --staged` compare la zone de staging avec le dernier commit.
- `git add -p` (optionnel) permet de valider par hunks (morceaux) pour des commits plus atomiques.

Partie 3 — Dépôt distant : remote, push, pull, clone

3.1 Créer le dépôt distant

Sur **GitHub** ou **GitLab** :

1. Créez un dépôt vide nommé `tp-git` (sans README initial pour éviter des divergences).
2. Copiez l'**URL HTTPS** du dépôt.

3.2 Lier le dépôt local au distant

$ git remote add origin https://<plateforme>/<utilisateur>/tp-git.git
$ git remote -v

Explications

- `origin` est l'alias par défaut du dépôt distant principal.
- `git remote -v` liste les URLs fetch/push.

Si votre branche locale s'appelle `master`, vous pouvez la renommer :

$ git branch -M main

3.3 Envoyer vos commits (push)

$ git push -u origin main

Explications

- `-u` définit `origin/main` comme **amont** (upstream) → les prochains `git push`/`git pull` sans arguments sauront où aller/venir.
- La première fois, une authentification via **token** personnel peut être demandée (GitHub) : suivez les instructions à l'écran.

3.4 Récupérer des changements (pull)

Si des modifications ont été faites sur la plateforme :

$ git pull

Explications

- `pull` = `fetch` + `merge` de l'amont vers votre branche locale.
- S'il n'y a pas de divergence → **fast-forward** (avance rapide). Sinon, un **merge commit** peut être créé.

3.5 Cloner un dépôt existant (clone)

$ cd ..
$ git clone https://<plateforme>/<utilisateur>/tp-git.git tp-git-clone
$ cd tp-git-clone

Explications

- `git clone` récupère l'historique et configure `origin` automatiquement.
- Optionnel : `--depth 1` pour un clone superficiel (sans historique complet).

À faire (exercice de synchro)

1. Dans **tp-git** (repo original), ajoutez `CONTRIBUTING.md`, commitez et poussez.
2. Dans **tp-git-clone** (clone), faites `git pull` → observez le nouveau fichier.
3. Modifiez `CONTRIBUTING.md` dans le clone, commitez, poussez.
4. Revenez dans **tp-git** et faites `git pull`.

Résultat attendu
Les deux dossiers contiennent les mêmes fichiers et l'historique converge après vos `push/pull`.

Partie 4 — Ignorer des fichiers avec `.gitignore`

4.1 Créer `.gitignore`

Dans la racine du dépôt :

```
# Fichiers temporaires / logs
*.tmp
*.log

# Python
__pycache__/
*.pyc

# Node / JS
node_modules/
```

```
dist/

# OS
.DS_Store
Thumbs.db
```

Explications

- `.gitignore` liste des **patterns** de fichiers à ne pas suivre.
- N'affecte que les fichiers **non encore suivis**. Pour arrêter de suivre un fichier déjà commité :

```
$ git rm -r --cached node_modules
$ echo "node_modules/" >> .gitignore
$ git commit -m "Ignore node_modules"
```

À faire

1. Ajoutez un `.gitignore` similaire.
2. Créez un fichier `debug.log`, vérifiez qu'il n'apparaît pas dans `git status`.
3. Poussez vos changements.

Résultat attendu
`debug.log` est ignoré et n'apparaît pas dans les fichiers non suivis.

Partie 5 — Branches et fusion (merge)

5.1 Créer et basculer de branche

Deux syntaxes équivalentes (préférez `git switch` si disponible) :

```
$ git switch -c dev        # crée et bascule sur dev
# ou
$ git checkout -b dev
```

À faire

1. Sur `dev`, modifiez `README.md` (ajoutez une section *Fonctionnalités*).
2. Commitez :

```
$ git add README.md
$ git commit -m "Ajoute section Fonctionnalités sur dev"
```

Résultat attendu
Un nouveau commit existe sur `dev`, absent de `main`.

5.2 Fusion simple (fast-forward)

$ git switch main
$ git merge dev

Explications

- Si `main` n'a pas divergé, Git avance simplement son pointeur (**fast-forward**).
- Supprimez la branche si vous n'en avez plus besoin : `git branch -d dev`.

5.3 Créer volontairement un petit conflit et le résoudre

À faire

1. Créez deux branches à partir de `main` : `dev-a` et `dev-b`.

$ git switch -c dev-a
$ echo "Ligne côté A" >> notes.txt
$ git add notes.txt && git commit -m "Ajoute ligne côté A"
$ git switch main
$ git switch -c dev-b
$ echo "Ligne côté B" >> notes.txt
$ git add notes.txt && git commit -m "Ajoute ligne côté B"

2. Fusionnez `dev-a` dans `main` :

$ git switch main
$ git merge dev-a

3. Tentez de fusionner `dev-b` dans `main` :

$ git merge dev-b

Résultat attendu
Git signale un **conflit** dans `notes.txt`. Ouvrez le fichier : vous voyez des marqueurs `<<<<<<<`, `=======`, `>>>>>>>`.

Résolution

- Éditez `notes.txt` pour garder le contenu souhaité (supprimez les marqueurs).
- Validez la résolution :

```
$ git add notes.txt
$ git commit   # message de merge par défaut
```

Alternative : `git merge --abort` pour annuler la fusion si nécessaire.

Partie 6 — Bonnes pratiques

- **Des commits petits et cohérents** : un commit = une idée/fonctionnalité/bugfix.
- **Messages clairs** (impératif, court) :
  - `Ajoute le formulaire d'inscription`
  - `Corrige l'erreur de validation email`
- **Pull régulier** avant de commencer une session de travail (`git pull`).
- **Branches par fonctionnalité** (`feature/…`, `fix/…`) ; fusion via **pull request** et relecture.
- **Ne commitez jamais** : secrets, mots de passe, clés privées, gros fichiers binaires (utilisez un coffre et/ou Git LFS si nécessaire).
- **.gitignore** dès le début du projet pour éviter le bruit.
- **Tags** pour marquer des versions (`git tag v1.0.0 && git push --tags`).
- **Garder l'historique lisible** : évitez les commits "WIP" sur `main`.

Annexe — Aide-mémoire rapide

Tâche	Commande
Configurer l'identité	`git config --global user.name "Nom"` · `git config --global user.email "email"`
Initialiser un dépôt	`git init`
État courant	`git status`
Ajouter au staging	`git add <fichier>` · `git add -p`
Commit	`git commit -m "Message"`
Historique concise	`git log --oneline --graph --decorate --all`
Comparer	`git diff` · `git diff --staged`
Lier un distant	`git remote add origin <URL>`

Pousser	`git push -u origin main`
Tirer	`git pull`
Cloner	`git clone <URL>`
Nouvelle branche	`git switch -c <branche>` (ou `git checkout -b <branche>`)
Fusionner	`git merge <branche>`
Supprimer une branche	`git branch -d <branche>`
Ignorer des fichiers	`.gitignore`
Arrêter de suivre	`git rm -r --cached <chemin>`