

Introduction à GIT

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2025-11-20

Table des matières

1 Objectifs pédagogiques	1
2 Pourquoi la Gestion de Versions est Importante	2
2.1 En pratique :	2
2.2 Qu'est-ce que Git ?	2
2.3 Différence entre Git et d'autres systèmes de gestion de versions	2
2.4 Pourquoi utiliser Git ?	2
3 Installation sur Différents Systèmes d'Exploitation	3
4 Configuration Initiale	3
5 Référentiels/Dépôts (Repositories)	3
5.1 Initialisation d'un Référentiel	3
6 Ajout de Fichiers au Suivi	3
7 Commits	4
7.1 Conseils pour rédiger des messages de commit utiles	4
7.2 Historique des Commits	5
8 Ignorer des fichiers	5
9 Affichage des Modifications	5
9.1 Affichage Amélioré de l'Histoire	6
10 Création et Gestion des Branches	6
10.1 Fusion (Merge) des Branches	7
11 Identifier et Résoudre des Conflits	7
12 Travailler avec des Référentiels Distants	8
12.1 Clonage d'un Référentiel	8
12.2 Pousser et Récupérer des Modifications	9
12.3 Ajout/Suppression de Remotes	10
13 Commandes supplémentaires Utiles	10
14 Git Cheatsheet	10

1 Objectifs pédagogiques

- Comprendre pourquoi utiliser un gestionnaire de versions.
- Savoir initialiser un dépôt, suivre des fichiers et créer des commits.
- Manipuler les branches, effectuer des merges et résoudre des conflits.

- Travailler avec un dépôt distant (clone, push, pull, remote).

Pré-requis : notions de base sur le système de fichiers et l'utilisation d'un terminal.

2 Pourquoi la Gestion de Versions est Importante

Gérer un projet de programmation sans gestionnaire de versions peut entraîner plusieurs problèmes :

- **S'envoyer l'entièreté du code par mail :**
 - Risque de confusion avec plusieurs versions du même fichier.
 - Difficulté à suivre les modifications apportées par chaque membre de l'équipe.
- **Partage de fichiers sur un serveur (Dropbox, etc.) :**
 - Problèmes de synchronisation.
 - Risque de perte de données si plusieurs personnes modifient le même fichier simultanément.
- **S'envoyer des patches :**
 - diff : Comparaison manuelle des différences entre deux fichiers/dossiers.

Pour résoudre ces problèmes, nous allons introduire Git, un gestionnaire de versions distribué qui permet de :

- Suivre les modifications de manière précise.
- Restaurer facilement des versions antérieures du projet.

2.1 En pratique :

- Comment revenir à une version antérieure ?
- Comment travailler à plusieurs sur un même projet ?
- Comment gérer des fonctionnalités expérimentales ?
- Avant Git : gestion manuelle des versions (copies, archives)

2.2 Qu'est-ce que Git ?

- **Définition :**
 - Git est un système de gestion de versions distribué.
- **Historique :**
 - Troisième génération de systèmes de contrôle de version :
 - Première génération : SCCS, RCS (années 70-80) : fichiers individuels, pas d'historique de projet
 - Deuxième génération : CVS, SVN (années 90-2000) : historique de projet, mais centralisé
 - Troisième génération : Git, Mercurial (années 2000+) : distribué, performant, branches légères

2.3 Différence entre Git et d'autres systèmes de gestion de versions

- **Centralisé vs. Distribué :**
 - Les systèmes centralisés (comme SVN) ont un serveur central unique.
 - Git est distribué, chaque développeur a une copie complète du référentiel.
- **Performance et Flexibilité :**
 - Git est conçu pour être rapide et efficace.
 - Permet de travailler hors ligne et de fusionner facilement les modifications.

2.4 Pourquoi utiliser Git ?

- **Gestion des Versions :**
 - Suivi précis des modifications apportées au code.
 - Possibilité de revenir à des versions antérieures.
- **Suivi des Modifications :**
 - Historique complet des modifications avec des messages de commit.
 - Identification facile des changements et des auteurs.
- **Collaboration Efficace :**
 - Permet à plusieurs développeurs de travailler simultanément sur le même projet.

- Facilite la fusion des modifications apportées par différents membres de l'équipe.
- **Travail Hors Ligne :**
 - Chaque développeur a une copie complète du référentiel.
 - Possibilité de travailler sans connexion internet et de synchroniser les modifications plus tard.

3 Installation sur Différents Systèmes d'Exploitation

- **Windows :**
 - Téléchargez l'installateur depuis git-scm.com.
 - Suivez les instructions de l'installateur.
 - Optionnel : utilisez Git Bash pour une interface en ligne de commande similaire à Unix.
 - WSL (Windows Subsystem for Linux) : installer une distribution Linux et utiliser Git via le terminal Linux.
- **macOS :**
 - Utilisez Homebrew : `brew install git`.
 - Téléchargez l'installateur depuis git-scm.com.
- **Linux :**
 - Utilisez le gestionnaire de paquets de votre distribution.
 - Exemple pour Debian/Ubuntu : `sudo apt-get install git`.
- **IDEs :**
 - La plupart des IDE modernes (VSCode, IntelliJ, PyCharm, etc.) intègrent Git.
 - Permet de gérer les versions directement depuis l'interface de l'IDE.

4 Configuration Initiale

- **Configurer le Nom d'Utilisateur :**
 - `git config --global user.name "Votre Nom"`
- **Configurer l'Adresse Email :**

- `git config --global user.email "votre.email@example.com"`

5 Référentiels/Dépôts (Repositories)

- **Dépôt (repository) :** emplacement où Git stocke les fichiers et l'historique des versions.
- **Local vs. Distant :**
 - Un référentiel local est stocké sur votre machine.
 - Un référentiel distant est hébergé sur un serveur (par exemple, GitHub). Bare repository : pas de copie de travail.

5.1 Initialisation d'un Référentiel

- **Créer un nouveau référentiel :**
 - `git init`
 - Initialise un nouveau référentiel Git dans le répertoire courant.

⚠ Exercice: Initialiser un dépôt local

Créez un répertoire de travail temporaire et initialisez-y un dépôt Git.

Solution

```
PROJECT_DIR=/tmp/github/ebpro/
notebook-git
```

```
cd ${PROJECT_DIR}
git init
```

```
Initialized empty Git repository
in /tmp/github/ebpro/notebook-
git/.git/
```

6 Ajout de Fichiers au Suivi

- **Ajouter des fichiers au suivi :**
 - `git add nom-du-fichier`

- Ajoute un fichier spécifique au suivi.
- `git add .` (`git add --all`)
 - Ajoute tous les fichiers modifiés au suivi.
- `git add -u`
 - stage modifications et suppressions mais pas les nouveaux fichiers.
- **Vérifier l'état :**
 - `git status`
 - Affiche l'état des fichiers dans le répertoire de travail et l'index et les actions possibles.

⚠ Exercice: Créer un fichier README et le mettre en suivi

Créez un fichier README contenant un titre et vérifiez l'état du dépôt avant et après le stage.

Solution

```
echo "# Hello" > README
git status
```

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to
  include in what will be committed)
  README

nothing added to commit but
untracked files present (use "git
add" to track)
```

```
git add README
git status
```

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:
  (use "git rm --cached <file>..."
  to unstage)
    new file:   README
```

7 Commits

• Qu'est-ce qu'un Commit ? :

- Un commit est un instantané des modifications apportées au projet.
- Chaque commit a un identifiant unique et un message descriptif.
 - Hash SHA-1 : identifiant unique (ex : `a1b2c3d4`)

• Messages de Commit :

- Les messages de commit doivent être clairs et concis.
- Exemple : `git commit -m "Ajout de la fonctionnalité X"`

• Tagger les Commits :

- Un tag est un pointeur immuable vers un commit spécifique, souvent utilisé pour marquer des versions (releases).
- `git tag nom-du-tag` : créer un tag léger.
- `git tag -a nom-du-tag -m "message"` : créer un tag annoté avec un message.

7.1 Conseils pour rédiger des messages de commit utiles

- Sujet (première ligne) : court et en impératif (<= 50 caractères). Exemple : Ajoute validation du formulaire.
- Séparez le sujet du corps par une ligne vide ; le corps explique *quoi*, *pourquoi* et *quel est l'impact* (ligne de 72 caractères max).
- Favorisez des commits atomiques : un changement conceptuel par commit.
- Si vous suivez la convention « Conventional Commits », utilisez un préfixe type(scope) : description (ex : `feat(auth): ajouter`

gestion du token). Types courants : feat, fix, docs, chore, refactor, test.

- Référencer issues/PR utiles dans le corps : Fixes #42 ou See #123.

```
feat(auth): ajouter gestion du token
d'expiration
```

Ajoute une vérification du token JWT côté client pour forcer le renouvellement automatique avant l'expiration. Cela évite des erreurs 401 inattendues pendant les sessions longues.

Fixes #42

⚠ Exercice: Valider les modifications avec un message de commit

Réalisez un commit décrivant brièvement la modification effectuée.

Solution

```
git commit -m "feat(README): add
basic documentation"
git status
```

```
[master (root-commit) 9a32adf]
feat(README): add basic
documentation
1 file changed, 1 insertion(+)
create mode 100644 README
On branch master
nothing to commit, working tree
clean
```

7.2 Historique des Commits

- Voir l'historique des commits :
 - git log
 - Affiche la liste des commits avec leurs messages, auteurs et dates.

```
git log
```

```
commit 9a32adf
Author: John Doe
<john.doe@nowhere.net>
Date: Thu Nov 20 23:15:06 2025
+0000
```

```
feat(README): add basic
documentation
```

8 Ignorer des fichiers

- Fichier .gitignore :
 - Permet de spécifier les fichiers et répertoires à ignorer par Git.
 - Utile pour éviter de suivre les fichiers temporaires, les dépendances, etc.

⚠ Exercice: Créer un fichier .gitignore pour ignorer les fichiers temporaires

Créez un fichier temporaire et ajoutez une règle dans .gitignore pour l'exclure du suivi.

Solution

```
touch bidon.tmp
echo "*.tmp">.gitignore
git status
git add .
git commit -m "feat(.gitignore): add
a .gitignore file to exclude
temporary files"
```

9 Affichage des Modifications

- Voir les différences :
 - git diff
 - Affiche les différences entre les fichiers modifiés et la dernière version validée.

```
echo "Bla bla" >> README
git diff
```

```
diff --git a/README b/README
index fec5601..9aeec5f 100644
--- a/README
+++ b/README
@@ -1,2 @@
 # Hello
+Bla bla
```

```
git add .
git commit -m "refactor(README):
update README content"
```

```
[master 53d47cc] refactor(README):
update README content
1 file changed, 1 insertion(+)
```

9.1 Affichage Amélioré de l'Historique

- Possibilité d'afficher l'historique de manière plus visuelle et informative.

```
alias lg="git log --graph --abbrev-
commit --decorate --
format=format:'%C(bold
blue)%h%C(reset) - %C(bold green)
(%ar)%C(reset) %C(white)%s%C(reset)
%C(dim white)-
%an%C(reset)%C(auto)%d%C(reset)' --
all"
```

```
lg
```

```
* 53d47cc - (0 seconds ago)
refactor(README): update README
content - John Doe (HEAD -> master)
* 10af630 - (1 second ago)
feat(.gitignore): add a .gitignore
file to exclude temporary files -
John Doe
* 9a32adf - (3 seconds ago)
```

```
feat(README): add basic
documentation - John Doe
```

10 Création et Gestion des Branches

- Les branches permettent de travailler sur différentes versions du projet simultanément.
- Commande pour créer une branche : `git branch nom-de-branche`
- Commande pour changer de branche : `git checkout nom-de-branche` (historique, très répandue).
 - Crée une nouvelle branche et bascule dessus :
 - `git checkout -b nom-de-branche` (historique).
 - Commandes modernes recommandées :
 - `git switch nom-de-branche` (Git 2.23+) pour changer de branche
 - `git switch -c nom-de-branche` pour créer + basculer.
 - `git restore <fichier>` pour restaurer un fichier depuis l'index ou depuis un commit.
- Une branche est un pointeur mobile vers un commit spécifique.
- Branche par défaut : `main` (ou `master`).
- `HEAD` : pointeur vers la branche courante ou un commit spécifique (détaché).

⚠ Exercice: Créer la branche `feature/main` et y basculer

Créez `feature/main`, ajoutez un fichier d'exemple et validez votre changement.

Solution

```
# commande recommandée (Git >= 2.23)
git switch -c feature/app
touch App.java
```

```
git add App.java
git commit -m "feat: ajouter
App.java"

# affichage de l'historique
(optionnel)
git log --oneline --graph --decorate
--all
```

```
Switched to a new branch 'feature/
app'
[feature/app ba42833] feat: ajouter
App.java
1 file changed, 0 insertions(+), 0
deletions(-)
create mode 100644 App.java
* ba42833 (HEAD -> feature/app)
feat: ajouter App.java
* 53d47cc (master) refactor(README):
update README content
* 10af630 feat(.gitignore): add
a .gitignore file to exclude
temporary files
* 9a32adf feat(README): add basic
documentation
```

10.1 Fusion (Merge) des Branches

- La fusion permet d'intégrer les modifications d'une branche dans une autre.
- Commande pour fusionner : `git merge nom-de-branche`
- fast-forward : si la branche cible n'a pas avancé, Git avance simplement le pointeur.

⚠ Exercice: Fusionner la branche `feature/main` dans `main`

Basculez sur la branche `main` et fusionnez la branche `feature/app` (créée précédemment) en conservant un commit de merge (`--no-ff`). Observez ensuite l'historique graphique pour vérifier que le merge a créé un commit dédié.

Solution

```
# assurez-vous d'être à jour et sur
main
git switch main
git fetch --all

# fusionner la branche feature/app
en forçant un commit de merge
git merge --no-ff feature/app -m
"merge: intégrer feature/app"

# vérifiez le graphe
git log --oneline --graph --decorate
--all
```

```
fatal: invalid reference: main
Already up to date.
* ba42833 (HEAD -> feature/app)
feat: ajouter App.java
* 53d47cc (master) refactor(README):
update README content
* 10af630 feat(.gitignore): add
a .gitignore file to exclude
temporary files
* 9a32adf feat(README): add basic
documentation
```

11 Identifier et Résoudre des Conflits

- Les conflits surviennent lorsque des modifications incompatibles sont apportées à la même partie d'un fichier dans différentes branches.
- Git signale les conflits lors de la fusion ou du rebase.
- Identifier les Conflits
 - Utiliser `git status` : Affiche les fichiers en conflit.
 - Utiliser `git diff` : Montre les différences entre les versions en conflit.
- Résoudre les Conflits
 - Édition Manuelle des Fichiers :

- Ouvrez les fichiers en conflit et modifiez-les pour résoudre les différences.
- Les sections en conflit sont marquées par <<<<<<, =====, et >>>>>>.
- Grâce à des outils de merge graphiques (ex : Meld, KDiff3, etc.) pour faciliter la résolution.
- ▶ **Valider après Résolution :**
 - Une fois les conflits résolus, ajoutez les fichiers modifiés avec `git add`.
 - Validez les modifications avec `git commit`.

12 Travailler avec des Référentiels Distants

- Remote repository : un dépôt hébergé sur un serveur distant (ex : GitHub, GitLab, Bitbucket).
- Permet la collaboration entre plusieurs développeurs.
- Commandes principales :
 - ▶ `git clone URL-du-référentiel` : cloner un dépôt distant.
 - ▶ `git push` : envoyer les modifications locales vers le dépôt distant.
 - ▶ `git fetch` : récupérer les modifications du dépôt distant sans les fusionner.
 - `git pull` : récupérer les modifications du dépôt distant et les fusionner avec le dépôt local. (équivalent de `git fetch + git merge`)

12.1 Clonage d'un Référentiel

- **Cloner un référentiel :**
 - ▶ `git clone URL-du-référentiel`
 - ▶ Crée une copie locale d'un référentiel distant.

```
git clone https://github.com/ebpro/HelloGit.git
```

```
Cloning into 'HelloGit'...
remote: Enumerating objects: 39,
done.
remote: Counting objects: 2%
(1/39)remote: Counting objects: 5%
(2/39)remote: Counting objects: 7%
(3/39)remote: Counting objects: 10%
(4/39)remote: Counting objects: 12%
(5/39)remote: Counting objects: 15%
(6/39)remote: Counting objects: 17%
(7/39)remote: Counting objects: 20%
(8/39)remote: Counting objects: 23%
(9/39)remote: Counting objects: 25%
(10/39)remote: Counting objects:
28% (11/39)remote: Counting objects:
30% (12/39)remote: Counting objects:
33% (13/39)remote: Counting objects:
35% (14/39)remote: Counting objects:
38% (15/39)remote: Counting objects:
41% (16/39)remote: Counting objects:
43% (17/39)remote: Counting objects:
46% (18/39)remote: Counting objects:
48% (19/39)remote: Counting objects:
51% (20/39)remote: Counting objects:
53% (21/39)remote: Counting objects:
56% (22/39)remote: Counting objects:
58% (23/39)remote: Counting objects:
61% (24/39)remote: Counting objects:
64% (25/39)remote: Counting objects:
66% (26/39)remote: Counting objects:
69% (27/39)remote: Counting objects:
71% (28/39)remote: Counting objects:
74% (29/39)remote: Counting objects:
76% (30/39)remote: Counting objects:
79% (31/39)remote: Counting objects:
82% (32/39)remote: Counting objects:
84% (33/39)remote: Counting objects:
87% (34/39)remote: Counting objects:
89% (35/39)remote: Counting objects:
92% (36/39)remote: Counting objects:
94% (37/39)remote: Counting objects:
97% (38/39)remote: Counting objects:
100% (39/39)remote: Counting
objects: 100% (39/39), done.
remote: Compressing objects: 4%
(1/21)remote: Compressing objects:
9% (2/21)remote: Compressing
objects: 14% (3/21)remote:
```



```

Compressing objects: 19%
(4/21)remote: Compressing objects:
23% (5/21)remote: Compressing
objects: 28% (6/21)remote:
Compressing objects: 33%
(7/21)remote: Compressing objects:
38% (8/21)remote: Compressing
objects: 42% (9/21)remote:
Compressing objects: 47%
(10/21)remote: Compressing objects:
52% (11/21)remote: Compressing
objects: 57% (12/21)remote:
Compressing objects: 61%
(13/21)remote: Compressing objects:
66% (14/21)remote: Compressing
objects: 71% (15/21)remote:
Compressing objects: 76%
(16/21)remote: Compressing objects:
80% (17/21)remote: Compressing
objects: 85% (18/21)remote:
Compressing objects: 90%
(19/21)remote: Compressing objects:
95% (20/21)remote: Compressing
objects: 100% (21/21)remote:
Compressing objects: 100% (21/21),
done.
Receiving objects: 2%
(1/39)Receiving objects: 5%
(2/39)Receiving objects: 7%
(3/39)Receiving objects: 10%
(4/39)Receiving objects: 12%
(5/39)Receiving objects: 15%
(6/39)Receiving objects: 17%
(7/39)Receiving objects: 20%
(8/39)Receiving objects: 23%
(9/39)Receiving objects: 25%
(10/39)remote: Total 39 (delta 3),
reused 38 (delta 2), pack-reused 0
(from 0)
Receiving objects: 28%
(11/39)Receiving objects: 30%
(12/39)Receiving objects: 33%
(13/39)Receiving objects: 35%
(14/39)Receiving objects: 38%
(15/39)Receiving objects: 41%
(16/39)Receiving objects: 43%
(17/39)Receiving objects: 46%
(18/39)Receiving objects: 48%

```

```

(19/39)Receiving objects: 51%
(20/39)Receiving objects: 53%
(21/39)Receiving objects: 56%
(22/39)Receiving objects: 58%
(23/39)Receiving objects: 61%
(24/39)Receiving objects: 64%
(25/39)Receiving objects: 66%
(26/39)Receiving objects: 69%
(27/39)Receiving objects: 71%
(28/39)Receiving objects: 74%
(29/39)Receiving objects: 76%
(30/39)Receiving objects: 79%
(31/39)Receiving objects: 82%
(32/39)Receiving objects: 84%
(33/39)Receiving objects: 87%
(34/39)Receiving objects: 89%
(35/39)Receiving objects: 92%
(36/39)Receiving objects: 94%
(37/39)Receiving objects: 97%
(38/39)Receiving objects: 100%
(39/39)Receiving objects: 100%
(39/39), 11.18 KiB | 602.00 KiB/s,
done.
Resolving deltas: 0%
(0/3)Resolving deltas: 33%
(1/3)Resolving deltas: 66%
(2/3)Resolving deltas: 100%
(3/3)Resolving deltas: 100% (3/3),
done.

```

12.2 Pousser et Récupérer des Modifications

- **Envoyer des modifications au référentiel distant :**
 - git push
 - Envoie les commits locaux au référentiel distant.
- **Mettre à jour le référentiel local avec les modifications distantes :**
 - git pull
 - Récupère les modifications du référentiel distant et les fusionne avec le référentiel local.
 - git pull --rebase

- garder un historique linéaire (reprend vos commits au-dessus des nouveaux commits distants).
- **Attention : --rebase réécrit l'historique local — ne l'utilisez pas sur des branches déjà partagées publiquement.**

12.3 Ajout/Suppression de Remotes

- Plusieurs référentiels distants peuvent être configurés pour un même projet.
 - Cela permet de travailler avec plusieurs remotes (ex : origin, upstream).
- **Ajouter un remote :**
 - `git remote add nom URL`
 - Ajoute un nouveau référentiel distant.
- **Supprimer un remote :**
 - `git remote remove nom`
 - Supprime un référentiel distant.
- **Lister les remotes :**
 - `git remote -v`
 - Affiche la liste des référentiels distants.

⚠ Exercice: Pousser un dépôt local vers un dépôt distant (GitHub)

Configurez un remote origin vers un dépôt GitHub et poussez votre branche locale.

Solution

```
# créer un dépôt distant sur GitHub
via l'UI puis :
git remote add origin
git@github.com:VOTRE_UTILISATEUR/
VOTRE_REPO.git
git push -u origin feature/main
```

13 Commandes supplémentaires Utiles

- `git stash` : Met de côté les modifications non validées.
- `git cherry-pick <commit>` : Applique un commit spécifique d'une autre branche.
- **Avertissement — commandes destructrices**
 - `git reset --hard <commit>` : replace HEAD et copie de travail — les modifications non committées sont perdues.
 - `git clean -fd` : supprime les fichiers non suivis (dossiers et fichiers) — utilisez `git clean -n` pour faire un essai.

14 Git Cheatsheet

- Initialiser / cloner
 - `git init` — initialiser un dépôt local.
 - `git clone <url>` — cloner un dépôt distant (ex : `git clone https://github.com/user/repo.git`).
- Stage / Commit / État
 - `git add -A` — stage toutes les modifications (ajouts, modifications, suppressions).
 - `git add -p` — sélectionner des morceaux (hunks) à stage.
 - `git commit -m "message"` — créer un commit avec un message concis.
 - `git commit --amend` — modifier le dernier commit (avant push).
 - `git status` — état du working tree et de l'index.
 - `git restore <fichier>` — annuler les modifications locales d'un fichier.
- Diff & Historique
 - `git diff` — différences non indexées.
 - `git diff --staged` — différences indexées vs HEAD.
 - `git log --oneline --graph --decorate --all` — vue compacte et visuelle de l'historique.

- Astuce : alias `lg` pour une vue enrichie (expliqué plus haut).
- Branches & navigation
 - `git branch` — lister les branches locales.
 - `git switch -c <branche>` — créer et basculer sur une branche (moderne; historique: `git checkout -b`).
 - `git switch <branche>` — basculer sur une branche (moderne; historique: `git checkout <branche>`).
 - `git rev-parse --abbrev-ref HEAD` — afficher le nom de la branche courante.
- Fusion & rebase
 - `git merge <branche>` — fusionner une branche dans la courante.
 - `git merge --no-ff <branche>` — garder un commit de merge explicite.
 - `git rebase <base>` — réappliquer les commits sur une autre base.
 - `git rebase -i <base>` — rebase interactif pour réorganiser/écrire l'historique local.
- Travailler avec les remotes
 - `git remote add origin <url>` — ajouter un remote (rm pour supprimer).
 - `git fetch` — récupérer les refs distantes sans fusionner.
 - `git pull --rebase` — récupérer puis rebaser vos commits au-dessus des commits distants (préférer pour branches locales; **attention**).
 - `git push -u origin <branche>` — pousser et définir l'upstream.
 - `git push --force-with-lease` — forcer un push en sécurité relative (préférer à `--force`).
- Sauvegarde temporaire & récupération
 - `git stash push -m "note"` / `git stash pop` — mettre de côté et restaurer des modifications non validées.
 - `git restore --source=<commit> <fichier>` — restaurer un fichier depuis un commit précis.
- `git reset --soft|--mixed|--hard <commit>` — déplacer HEAD (danger : `--hard` détruit les modifications non committées).
- `git reflog` — retrouver des références perdues après des opérations destructrices.
- Tags et utilitaires
 - `git tag -a v1.0 -m "message"` — créer une tag annotée.
 - `git cherry-pick <commit>` — appliquer un commit spécifique sur la branche courante.
 - `git bisect start` / `git bisect good|bad` — bisect pour trouver le commit fautif.