

Introduction à GIT

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

2026-10-14

Table des matières

1	Entrepot du cours	1
2	Pourquoi la Gestion de Versions est Importante	1
3	Transition vers Git	2
4	En pratique :	2
5	Qu'est-ce que Git ?	2
5.1	Différence entre Git et d'autres systèmes de gestion de versions	2
5.2	Pourquoi utiliser Git ?	2
6	Installation sur Différents Systèmes d'Exploitation	3
7	Configuration Initiale	3
8	Référentiels/Dépôts (Repositories)	3
9	Initialisation d'un Référentiel	3
9.1	Pratique : Initialisation d'un Référentiel	3
10	Ajout de Fichiers au Suivi	4
10.1	Exercice Pratique : Ajout de Fichiers au Suivi	4
11	Commits	4
11.1	Exercice Pratique : Création d'un Commit	4
12	Historique des Commits	5
12.1	Affichage Amélioré de l'Historique ..	5
13	Ignorer des fichiers	5
13.1	Exercice Pratique : Création d'un Fichier .gitignore	5
14	Affichage des Modifications	6
15	Création et Gestion des Branches	6

15.1	Exercice Pratique : Création et Changement de Branches	6
15.2	Fusion (Merge) des Branches	7
16	Travailler avec des Référentiels Distants ..	7
16.1	Clonage d'un Référentiel	7
16.2	Pousser des Modifications	7
16.3	Récupérer des Modifications	7
16.4	Ajout/Suppression de Remotes	7
16.5	Exercice Pratique : Clonage et Push	7
17	Gestion des Branches et des Fusions	8
17.1	Exercices Pratiques Branches et Fusions	8
18	Résolution des Conflits	8
18.1	Identifier les Conflits	8
18.2	Résoudre les Conflits	8
19	Commandes supplémentaires Utiles	8
20	Résumé des Points Clés	8
21	Git Cheatsheet	8
22	Bonnes pratiques rapides	9

1 Entrepot du cours

Git Repository Status

△ Status | Not a git repository |

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.
 - **patch** : Application manuelle des modifications, source d'erreurs.
- Créé par Linus Torvalds en 2005.
- **Historique :**
 - Développé pour gérer le code source du noyau Linux.
 - 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

3 Transition vers Git

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

- Suivre les modifications de manière précise.
- Collaborer efficacement avec d'autres développeurs.
- Restaurer facilement des versions antérieures du projet.

4 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)

5 Qu'est-ce que Git ?

- **Définition :**
 - Git est un système de gestion de versions distribué.

5.1 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.

5.2 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.

6 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.

7 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"`

8 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.

9 Initialisation d'un Référentiel

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

9.1 Pratique : Initialisation d'un Référentiel

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

```
hint: Using 'master' as the name
hint: for the initial branch. This default
hint: branch name
hint: is subject to change. To
hint: configure the initial branch name to
hint: use in all
hint: of your new repositories, which
hint: will suppress this warning, call:
hint:
hint:     git config --global
hint:     init.defaultBranch <name>
hint:
hint: Names commonly chosen instead
```

```
of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created
branch can be renamed via this
command:
hint:
hint: git branch -m <name>
hint:
hint: Disable this message with "git
config set advice.defaultBranchName
false"
Initialized empty Git repository in /
tmp/github/ebpro/notebook-git/.git/
```

10 Ajout de Fichiers au Suivi

- **Ajouter des fichiers au suivi :**
 - `git add nom-du-fichier`
 - Ajoute un fichier spécifique au suivi.
 - `git add .`
 - Ajoute tous les fichiers modifiés au suivi.
 - `git status`
 - Indique l'état courant et les actions possibles.
- **Vérifier l'état :**
 - `git status`
 - Affiche l'état des fichiers dans le répertoire de travail et l'index.

10.1 Exercice Pratique : Ajout de Fichiers au Suivi

- Créez un fichier README et ajoutez-y du contenu.
- Ajoutez le fichier au suivi avec `git add`.

```
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

11 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 :
alb2c3d4)
- **Messages de Commit :**
 - Les messages de commit doivent être clairs et concis.
 - Exemple : `git commit -m "Ajout de la fonctionnalité X"`

11.1 Exercice Pratique : Création d'un Commit

- Validez les modifications avec un message de commit.

```
git commit -m "Ajout d'une documentation de base"
```

```
[master (root-commit) 6da78f5] Ajout d'une documentation de base
1 file changed, 1 insertion(+)
create mode 100644 README
```

```
git status
```

```
On branch master
nothing to commit, working tree clean
```

12 Historique des Commits

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

```
git log
```

```
commit 6da78f5579d7b57b71067d1a7a77e1e6f7
(HEAD -> master)
Author: John Doe <john.doe@nowhere.net>
Date: Wed Oct 14 14:32:06 2026 +0000
```

```
Ajout d'une documentation de base
```

12.1 Affichage Amélioré de l'Histoire

- 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
```

```
* 6da78f5 - (10 seconds ago) Ajout d'une documentation de base - John Doe (HEAD -> master)
```

13 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.

13.1 Exercice Pratique : Création d'un Fichier .gitignore

- Créez un fichier temporaire (extension .tmp) et ajoutez-le au .gitignore.

```
touch bidon.tmp
echo "*.tmp">.gitignore
git status
git add .
git commit -m "ajoute un gitignore."
```

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
.gitignore
```

```
nothing added to commit but untracked files present (use "git add" to track)
[master ac8140c] ajoute un gitignore.
```

```
1 file changed, 1 insertion(+)
create mode 100644 .gitignore
```

14 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..9aee5f 100644
--- a/README
+++ b/README
@@ -1,2 @@
 # Hello
+Bla bla
```

```
git add .
git commit -m "améliore la description"
```

```
[master a39d866] améliore la description
1 file changed, 1 insertion(+)
```

15 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`
 - Crée une nouvelle branche et bascule dessus : `git checkout -b nom-de-branche`

- `switch` : `git switch nom-de-branche` (Git 2.23+), `git switch -c nom-de-branche` (crée et switch), nouvelle commande plus intuitive.

15.1 Exercice Pratique : Création et Changement de Branches

- Créez une nouvelle branche nommée `feature/main` et basculez dessus.
- Ajoutez un fichier `Main.java` et validez les modifications.

```
git branch feature/main
git checkout feature/main
touch Main.java
git add .
```

```
Switched to branch 'feature/main'
```

```
lg
```

```
* a39d866 - (1 second ago) améliore la description - John Doe (HEAD -> feature/main, master)
* ac8140c - (1 second ago) ajoute un gitignore. - John Doe
* 6da78f5 - (12 seconds ago) Ajout d'une documentation de base - John Doe
```

```
lg1
```

```
* a39d866 - Wed, 14 Oct 2026 14:32:17 +0000 (1 second ago) (HEAD -> feature/main, master)
|      améliore la description - John Doe
* ac8140c - Wed, 14 Oct 2026 14:32:17 +0000 (1 second ago)
|      ajoute un gitignore. -
```

```
John Doe
* 6da78f5 - Wed, 14 Oct 2026 14:32:06
+0000 (12 seconds ago)
    Ajout d'une documentation
de base - John Doe
```

15.2 Fusion (Merge) des Branches

- La fusion permet d'intégrer les modifications d'une branche dans une autre.
- Commande pour fusionner : `git merge nom-de-branch`

16 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`)

16.1 Clonage d'un Référentiel

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

16.2 Pousser des Modifications

- **Envoyer des modifications au référentiel distant :**
 - `git push`

- Envoie les commits locaux au référentiel distant.

16.3 Récupérer des Modifications

- **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.

16.4 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.

16.5 Exercice Pratique : Clonage et Push

1. Poussez un projet local vers un nouveau dépôt distant.
 - Créez un nouveau dépôt sur GitHub.
 - Ajouter un remote `origin` pointant vers ce dépôt.
 - Poussez votre branche `feature/main` vers ce dépôt.
2. Créer un nouveau repository GitHub MyJavaProject
 - Créer un projet Java dans ce repository

17 Gestion des Branches et des Fusions

- Les branches permettent de travailler sur différentes versions du projet simultanément.
- La fusion permet d'intégrer les modifications d'une branche dans une autre.
- Commandes principales :
 - `git branch nom-de-branche` : créer une nouvelle branche.
 - `git checkout nom-de-branche` : changer de branche. (ou `git switch nom-de-branche` (Git 2.23+))
 - `git merge nom-de-branche` : fusionner une branche dans la branche courante.

17.1 Exercices Pratiques Branches et Fusions

1. Créer une nouvelle branche `feature/login` à partir de `main`.
2. Basculer vers la branche `feature/login`.
3. Ajouter un fichier `Login.java` et valider les modifications.
4. Revenir à la branche `main`.
5. Fusionner la branche `feature/login` dans `main`.

18 Résolution 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 rebasage.
- **Résolution des Conflits :**
 - Les conflits doivent être résolus manuellement avant de pouvoir valider les modifications.

18.1 Identifier les Conflits

- Utiliser `git status` :
 - Affiche les fichiers en conflit.
- Utiliser `git diff` :

- Montre les différences entre les versions en conflit.

18.2 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`.

19 Commandes supplémentaires Utiles

- `git stash` : Met de côté les modifications non validées.
- `git cherry-pick <commit>` : Applique un commit spécifique d'une autre branche.
- `git rebase` : Rejoue les commits sur une autre base.

20 Résumé des Points Clés

- **Importance de Git :**
 - Gestion efficace des versions.
 - Collaboration facilitée.
 - Suivi précis des modifications.

21 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`).
- Ajouter / valider / état
 - `git add <fichier|.>` — stage des modifications (`git add -A` pour tout).

- `git commit -m "message"` — créer un commit (message clair).
- `git status` — voir l'état du working tree.
- `git diff` — voir les différences non indexées.
- `git diff --staged` — différences entre index et dernier commit.
- Historique et logs
 - `git log --oneline --graph --decorate --all` — vue compacte et visuelle.
 - Astuce : alias `lg` (voir plus haut dans le cours) pour un log amélioré.
- Branches & navigation
 - `git branch` — lister les branches locales.
 - `git switch -c <branche>` — créer et basculer sur une branche.
 - `git switch <branche>` ou `git checkout <branche>` — changer de branche.
- Fusion & rebase
 - `git merge <branche>` — fusionner une branche dans la courante (ex : `--no-ff` pour garder l'historique).
 - `git rebase <branche>` / `git rebase -i <base>` — réappliquer des commits (interactive pour nettoyer l'historique).
- Travailler avec le distant
 - `git remote add origin <url>` — ajouter un remote.
 - `git fetch` — récupérer les refs distantes sans fusionner.
 - `git pull --rebase` — récupérer et réappliquer vos commits au-dessus des nouveaux commits distants.
 - `git push -u origin <branche>` — pousser et définir upstream.
- Sauvegarde temporaire & récupération
 - `git stash push -m "note"` / `git stash pop` — mettre de côté des modifications.
 - `git reset --soft|--mixed|--hard <commit>` — déplacer HEAD (attention avec `--hard`).
 - `git restore <fichier>` — restaurer un fichier depuis l'index ou un commit.
- Autres outils utiles
 - `git tag <name>` — marquer une version.
 - `git cherry-pick <commit>` — appliquer un commit spécifique ailleurs.
 - `git bisect` — débogage binaire pour trouver un commit fautif.
- .gitignore (exemples)
 - Ignorer fichiers temporaires et binaires :


```

*.log
__pycache__/
node_modules/
*.class
          
```
 - Astuce : `git status --ignored` pour voir les fichiers ignorés.

22 Bonnes pratiques rapides

- Commits petits et atomiques.
- Messages explicites (quoi et pourquoi).
- Pousser régulièrement et puller avant de commencer à travailler.
- Utiliser des branches pour les fonctionnalités (feature/*) et des PRs pour la revue.