ATHENA - Formation GIT Débutant

Alain Piallat

13 janvier 2026

Sommaire

Cette formation couvre à la fois les notions fondamentales et des thèmes avancés. Pour faciliter l'apprentissage, les diapositives présentant des concepts élémentaires - nécessaires pour une utilisation basique de Git - seront signalées par le symbole ♣ dans leur titre.

Qu'est-ce que GIT ?

- Créé par Linus Torvalds en 2005 pour le développement du noyau Linux.
- GIT est un système de contrôle de version.
- Il permet de suivre les modifications apportées aux fichiers et de collaborer avec d'autres utilisateurs.
- GIT est distribué, chaque utilisateur possède une copie complète du dépôt.
- Utilisé principalement pour le code source, mais peut gérer tout type de fichier.

Git vs GitHub/GitLab

Git :

- Système de contrôle de version
- Fonctionne en local et en ligne de commande
- Gère les versions, branches, fusions, etc.

GitHub / GitLab :

- Plateformes d'hébergement de dépôts Git
- Fournissent une interface web
- Facilitent la collaboration et le partage de code
- Offrent des fonctionnalités supplémentaires (pull requests, wikis, etc.)

Git est l'outil, GitHub est le service en ligne basé sur Git.

BDTech

Installation ♣

Linux :

- Ubuntu/Debian : `sudo apt install git`
- Fedora : `sudo dnf install git`

Windows et macOS :

- Aller sur https://git-scm.com/install/
- Suivre les instructions pour votre OS

Vérifier l'installation :

```
1 user@machine:~$ git --version
2 git version 2.x.x
```

Configuration initiale ♣

Configurer votre identité (obligatoire) :

```
1 git config --global user.name "Votre Nom"
2 git config --global user.email "email@example.com"
```

Vérifier la configuration :

```
1 user@machine:~$ git config --list
2 user.name=Votre Nom
3 user.email=email@example.com
```

Il existe trois niveaux de configuration : Pour tout les utilisateurs (--system), pour l'utilisateur courant (--global), pour le dépôt courant (sans option ou --local).

Autres options de configuration courantes

Configurer l'éditeur par défaut :

```
1 git config --global core.editor <commande-editeur>
```

Exemples : `code -wait` (VSCode), `nano`, `vim`

Gérer les fins de ligne :

```
1 git config --global core.autocrlf true
```

Permet d'éviter les problèmes de fin de ligne entre Windows et Linux/macOS.

Il existe de nombreuses autres options qui ne seront pas abordées ici.
https://git-scm.com/book/fr/v2/Personnalisation-de-Git-Configuration-de-Git

HTTPS vs SSH ♣

Pour interagir avec des dépôts git distants (push, pull), deux méthodes principales existent HTTPS et SSH.

HTTPS :

- Simple à configurer
- Nécessite de saisir vos identifiants à chaque interaction
- Moins sécurisé pour les opérations fréquentes

SSH (recommandé) :

- Plus sécurisé et pratique
- Utilise des clés SSH pour l'authentification
- Nécessite une configuration initiale (mais plus rien par la suite)

Configuration SSH - Génération de clés ♣

Étape 1 : Générer une paire de clés SSH

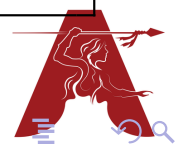
```
ssh-keygen -t ed25519 -C "votre.email@ensea.fr"
```

- Appuyez sur Entrée pour accepter l'emplacement par défaut
- Optionnel : entrez une passphrase pour plus de sécurité

Étape 2 : Démarrer l'agent SSH

```
# Linux
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519

# Windows (PowerShell)
Get-Service -Name ssh-agent | Set-Service -StartupType
    Manual # Necessite les droits admin
Start-Service ssh-agent
ssh-add ~\.ssh\id_ed25519
```

BDTech



Configuration SSH - Ajout sur GitHub ♣

Étape 3 : Copier la clé publique

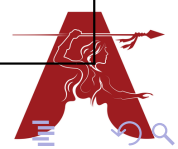
```
# Linux
cat ~/.ssh/id_ed25519.pub

# Windows (PowerShell)
Get-Content ~\.ssh\id_ed25519.pub
```

Étape 4 : Ajouter la clé sur GitHub/GitLab

1. Aller dans Settings > SSH and GPG keys
2. Cliquer sur "New SSH key"
3. Coller votre clé publique
4. Donner un titre descriptif (ex : "PC personnel")
5. Cliquer sur "Add SSH key"

Tester la connexion :

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



Plomberie vs Porcelaine

Plomberie (plumbing) :

- Commandes basses niveaux
- Manipulation directe des objets Git
- Exemples : `git hash-object`, `git cat-file`

Porcelaine (porcelain) :

- Commandes haut niveau
- Interfaces conviviales pour les utilisateurs
- Exemples : `git add`, `git commit`, `git push`

Nous nous concentrerons sur la porcelaine dans cette formation.

BDTech

Architecture de Git

Principes clés :

- Chaque dépôt est une base de données complète
- Les modifications sont enregistrées via des commits
- Le staging permet de préparer les changements avant le commit

Trois zones principales :

1. **Working Directory** : vos fichiers de travail
2. **Staging Area (Index)** : zone de préparation
3. **Repository (.git)** : historique des versions

$$\texttt{Working Dir} \xrightarrow{\textit{git add}} \texttt{Staging} \xrightarrow{\textit{git commit}} \texttt{Repository}$$

Local vs Remote

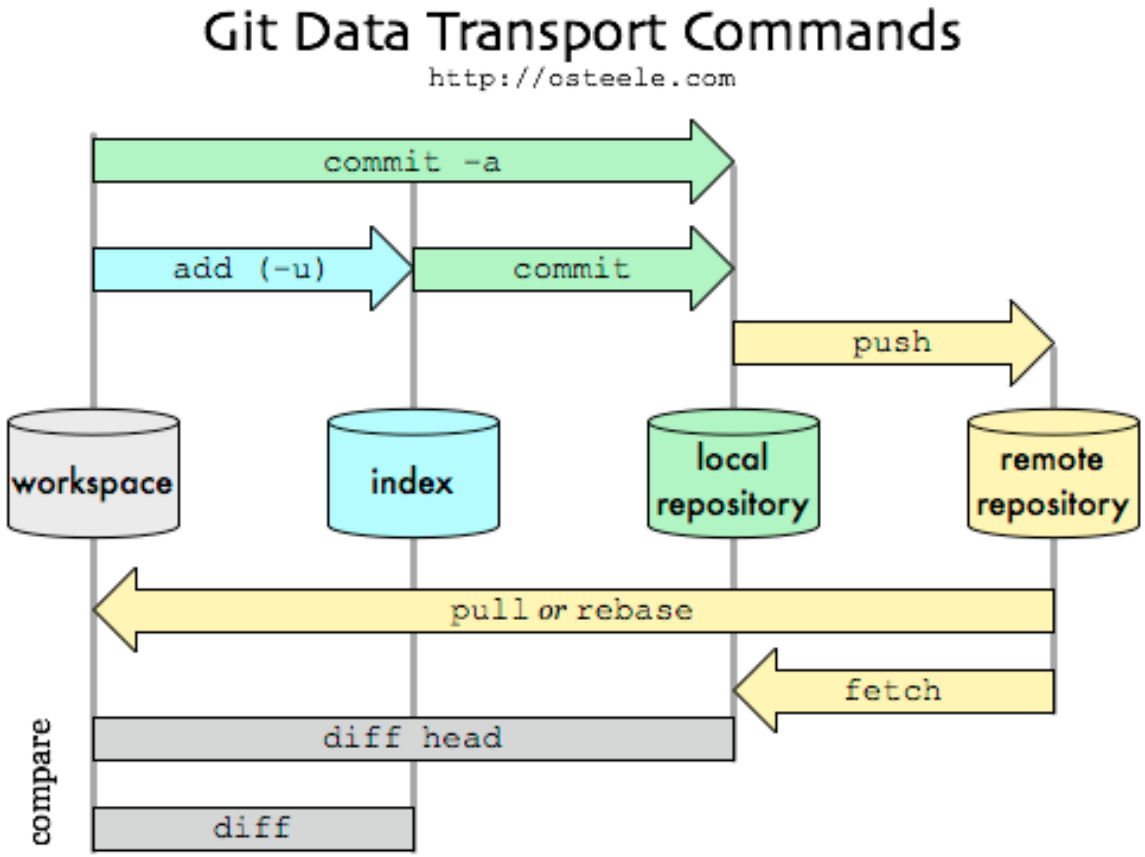
Dépôt local :

- Sur votre machine
- Travail hors ligne possible
- Commits locaux rapides

Dépôt distant (remote) :

- Hébergé sur GitHub/GitLab/etc.
- Partage avec l'équipe
- Sauvegarde externe

Pricipales commandes essentielles à maîtriser ♣



git init / git clone ♣

Créer un nouveau dépôt :

```
1 git init [nom-du-dossier]
```

Cloner un dépôt existant :

```
1 git clone https://github.com/user/repo.git [nom]
2 git clone git@github.com:user/repo.git [nom]
```

Le clonage crée automatiquement un dépôt local avec un remote nommé "origin"

Attention : l'url est modifiée avec ssh

git add ♣

Ajouter des fichiers au staging :

```
1 git add fichier.txt        # Un fichier specifique
2 git add dossier/           # Tout un dossier
3 git add .                  # Tous les fichiers modifies
4 git add *.py               # Tous les fichiers .py
```

Le staging permet de :

- Choisir quels changements inclure dans le commit
- Organiser ses commits de manière logique
- Vérifier avant de valider définitivement

git commit ♣

Enregistrer les modifications :

```
1 git commit -m "Message descriptif"
```

Commit avec éditeur pour message long :

```
1 git commit
```

Ajouter et commiter en une fois :

```
1 git commit -a # Seulement fichiers suivis
```

Chaque commit crée un point de sauvegarde unique

git log

Visualiser l'historique :

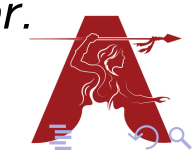
```
1 git log                          # Historique complet
2 git log --oneline                # Version condensee
3 git log --graph                  # Avec representation
      graphique
4 git log --all --decorate         # Toutes les branches
```

Affiche pour chaque commit :

- Hash unique (identifiant)
- Auteur et date
- Message de commit

--oneline et --graph peuvent être combinés, cela donne une vue d'ensemble claire de l'historique (similaire à celle fournie par des outils graphiques).
git log ouvre vim par défaut pour la navigation. Taper 'q' pour quitter.

BDTech



git diff

Voir les modifications :

```
1 git diff                          # Modifications non stagees
2 git diff --staged                 # Modifications stagees
3 git diff HEAD                     # Toutes les modifications
4 git diff commit1 commit2          # Entre deux commits
```

Utile pour :

- Vérifier ce qui a changé avant de commiter
- Comprendre les différences entre versions
- Relire son code avant validation

git rm et git mv

Supprimer un fichier du dépôt :

```
1 git rm fichier.txt                # Supprime et stage
2 git rm --cached fichier.txt       # Retire du suivi Git
      uniquement
```

Renommer ou déplacer un fichier :

```
1 git mv ancien.txt nouveau.txt
2 git mv fichier.txt dossier/
```

Ces commandes stagent automatiquement les changements

git status

Voir l'état du dépôt :

```
1 git status
```

Affiche :

- La branche courante
- Les fichiers modifiés
- Les fichiers en staging
- Les fichiers non suivis

Commande la plus utilisée ! À utiliser sans modération.

git status example

```
1  On branch main
2  Changes to be committed:
3    (use "git restore --staged <file>..." to unstage)
4          modified:   fichier1.txt
5          new file:   fichier2.txt
6  Changes not staged for commit:
7    (use "git add <file>..." to update what will be
       committed)
8    (use "git restore <file>..." to discard changes in
       working directory)
9          modified:   fichier3.txt
10 Untracked files:
11   (use "git add <file>..." to include in what will be
       committed)
12         fichier4.txt
```

git push ♣

Envoyer les commits vers le dépôt distant :

```
1 git push                              # Push la branche
      courante
2 git push <remote> <branch>    # Push vers la branche de
       remote
```

Important :

- Partage vos modifications avec l'équipe
- Nécessite des droits d'écriture
- Sauvegarde externe de votre travail

git pull ♣

Récupérer les modifications distantes :

```
1 git pull                          # Pull la branche
      courante
2 git pull origin main              # Pull depuis main sur
      origin
```

Git pull = git fetch + git merge

- Récupère les changements distants
- Fusionne avec votre branche locale dans votre working directory
- Peut provoquer des conflits à résoudre manuellement

Toujours pull avant de commencer à travailler !

Quand et comment commiter

Principe d'atomicité :

- Un commit = une modification logique
- Pas trop gros (difficile à relire)
- Pas trop petit (historique pollué)

Fréquence :

- Après chaque fonctionnalité complète
- Avant de changer de tâche

Syntaxe des messages de commit

Format standard :

```
Type: Sujet court (50 caracteres max)

Description optionnelle plus detaillee expliquant
le pourquoi et le comment du changement.
```

Types courants :

- `feat:` nouvelle fonctionnalité
- `fix:` correction de bug
- `docs:` documentation

Exemples de bons messages

Bons exemples :

```
feat: Ajoute l'authentification par OAuth
fix: Corrige le bug de connexion sur Safari
docs: Met a jour le README avec instructions Docker
```

Mauvais exemples :

```
update                  # Trop vague
fixed stuff             # Pas assez descriptif
azertyuiop              # Inutile
WIP                     # Work In Progress, a eviter
```

Le fichier .gitignore

Que faut-il exclure du versioning ?

- Fichiers générés (build, compiled)
- Dépendances (`node_modules/`, `venv/`)
- Fichiers de configuration locale
- Fichiers sensibles (clés API, mots de passe)
- Fichiers temporaires, logs
- Fichiers IDE (`.vscode/`, `.idea/`)

Ressource : `github.com/github/gitignore`

Exemple de .gitignore

```
# Fichiers de compilation
*.o
*.s

# Fichiers IDE
.vscode/

# Fichiers sensibles
.env
config.local.json
```

README.md

Le README est la vitrine de votre projet
Contenu typique :

- Titre et description du projet
- Instructions d'installation
- Exemples d'utilisation
- Technologies utilisées
- Comment contribuer
- Licence
- Contact/auteurs

Privilégier le format Markdown

Choisir une licence

Pourquoi une licence ?

- Définit les droits d'utilisation
- Protège votre travail
- Clarifie les conditions de partage

Licences courantes :

- **MIT :** très permissive, usage libre
- **GPL :** copyleft, modifications open-source
- **Apache 2.0 :** permissive avec protection brevets
- **CC BY :** pour contenu non-logiciel

Ressource : `choosealicense.com`

Principe et utilité des branches

Une branche = une ligne de développement parallèle
Cas d'usage :

- Développer une nouvelle fonctionnalité
- Corriger un bug sans perturber le code principal
- Expérimenter sans risque
- Travailler à plusieurs sans conflit

git branch ♣

Lister les branches :

```
git branch        # Branches locales
git branch -a     # Toutes les branches (+ remotes)
git branch -v     # Avec dernier commit
```

Créer une branche :

```
git branch nom-branche
```

Supprimer une branche :

```
git branch -d nom-branche      # Si fusionnee
git branch -D nom-branche      # Force la suppression
```

BDTech

git checkout / git switch

Changer de branche (ancienne méthode) :

```
1 git checkout nom-branche
2 git checkout -b nouvelle-branche   # Creer et basculer
```

Nouvelle syntaxe (Git 2.23+) :

```
1 git switch nom-branche
2 git switch -c nouvelle-branche     # Creer et basculer
```

git switch est plus clair et recommandé

git merge ♣

Fusionner une branche dans la branche courante :

```
1 git switch main
2 git merge feature-xyz
```

Types de merge :

- **Fast-forward :** simple avance du pointeur
- **Merge commit :** crée un commit de fusion (en cas de conflit)
- **Squash merge :** regroupe tous les commits en un seul

```
1 git merge --squash feature-xyz
```

Résolution de conflits

Un conflit survient quand :

- Deux branches modifient les mêmes lignes
- Git ne peut pas fusionner automatiquement

Résolution :

1. Git marque les conflits dans les fichiers
2. Ouvrir les fichiers et choisir la bonne version
3. Supprimer les marqueurs de conflit
4. `git add` les fichiers résolus
5. `git commit` pour finaliser le merge

Les IDE modernes facilitent grandement cette étape

Marqueurs de conflit

Exemple de conflit :

```
<<<<<<< HEAD
Code de votre branche courante
=======
Code de la branche a fusionner
>>>>>>> feature-xyz
```

Après résolution :

```
Code final que vous avez choisi
```

git rebase : réorganiser l'historique

Rebase applique vos commits sur une autre base :

```
1 git switch feature
2 git rebase main
```

Différence avec merge :

- Merge : conserve l'historique complet
- Rebase : crée un historique linéaire plus propre

Attention : ne jamais rebaser des commits déjà pushés !

Commandes avancés

Dans cette section, nous allons abordées des commandes avancées pouvant causer des modifications importantes dans votre dépôt voir la perte de données si elles sont mal utilisées.

Renseignez-vous bien avant de les utiliser.

git fetch : récupérer sans fusionner

Récupérer les modifications distantes sans fusion :

```
1 git fetch                  # Recupere tous les remotes
2 git fetch origin main      # Recupere main depuis origin
```

Comparer avec votre branche locale :

```
1 git diff main origin/main
```

Utile pour voir les changements avant de fusionner

git stash : sauvegarder temporairement

Mettre de côté des modifications :

```
1 git stash                          # Sauvegarde et nettoie
2 git stash save "Message"           # Avec description
```

Récupérer les modifications :

```
1 git stash list                          # Lister les stashs
2 git stash apply [stash@{index}]         # Applique le dernier
3 git stash pop                           # Applique et supprime
4 git stash drop [stash@{index}]          # Supprime un stash
```

Utile pour : changer de branche rapidement sans commiter

git reset : annuler des modifications

Cette commande permet de réécrire l'historique en annulant des commits.

Trois modes de reset :

```
1 git reset --soft HEAD~1      # Garde staging et working
2 git reset --mixed HEAD~1     # Garde working (defaut)
3 git reset --hard HEAD~1      # Supprime tout
```

Retirer du staging :

```
1 git reset fichier.txt
```

`HEAD` désigne le commit courant. le `1` signifie "le parent", `2` le grand-parent, etc.

Attention : `-hard` est destructif !

git revert : annuler proprement

Créer un commit qui annule un commit précédent :

```
1 git revert abc123          # Annule le commit abc123
2 git revert HEAD            # Annule le dernier commit
```

Différence avec reset :

- **reset :** réécrit l'historique (fait disparaître des commits)
- **revert :** ajoute un nouveau commit (plus transparent)

Préférer revert pour des commits déjà pushés

git cherry-pick

Appliquer un commit spécifique sur la branche courante :

```
1 git cherry-pick abc123
```

Utile pour :

- Récupérer un fix d'une autre branche
- Appliquer sélectivement des commits
- Éviter un merge complet

Workflow collaboratif

Dans cette section, nous allons sortir du cadre de git pour aborder les dépots collaboratifs hébergés sur des plateformes comme GitHub ou GitLab.

Fork : copier un projet

Le fork crée une copie indépendante d'un dépôt. Il permet de travailler sur un projet sans avoir besoins d'être ajouté en tant que collaborateur.

Processus :

1. Fork sur GitHub/GitLab (bouton "Fork")
2. Clone de votre fork en local
3. Modifications et commits
4. Push vers votre fork
5. Pull request vers le projet original

Pull Requests / Merge Requests

Pull Request ou PR (GitHub) / Merge Request (GitLab) : demande de fusionner vos modifications dans le dépôt principal. Elle permet :

- Revue de code avant intégration
- Discussion sur les modifications
- Traçabilité des changements
- Validation par l'équipe

Standard pour le développement collaboratif moderne

Revue de code

La revue de code consiste à examiner les modifications proposées dans une pull request et à fournir des retours constructifs. Elle peut inclure :

- Vérification de la qualité du code
- Suggestions d'amélioration
- Détection de bugs potentiels
- Validation des normes de codage
- Tests et vérifications

Erreurs courantes

Dans cette section, nous allons aborder quelques erreurs courantes rencontrées lors de l'utilisation de Git ainsi que leurs solutions.

Commit sur la mauvaise branche

Problème : vous avez commité sur `main` au lieu de votre branche
Solution (commit non pushé) :

```
# Annule le commit (en gardant les changements)
git reset --soft HEAD~1
git switch -c feature          # Cree la bonne branche
git commit -m "Message"        # Recommite
```

Ou avec cherry-pick :

```
git switch -c feature          # Nouvelle branche
git cherry-pick main           # Copie le commit
git switch main
git reset --hard HEAD~1        # Supprime sur main
```

Conflit de merge bloquant

Vous êtes bloqué en plein merge avec des conflits
Options :

```
# Resoudre les conflits manuellement
# puis :
git add fichiers-resolus
git commit

# Ou abandonner le merge :
git merge --abort

# Ou avec rebase :
git rebase --abort
```

Où trouver de l'aide ?

Ressources utiles :

- Documentation officielle : git-scm.com
- `git help <commande>` dans le terminal
- GitHub Learning Lab
- Stack Overflow
- Pro Git (livre gratuit : git-scm.com/book/en/v2)

Interfaces graphiques

Il existe de nombreuses interfaces graphiques pour Git, facilitant son utilisation comme `Git GUI` qui est fourni avec Git. Par ailleurs, la plupart des IDE intègrent Git nativement (VSCode, IntelliJ, etc.)

Documentation et tutoriels

Sites recommandés :

- git-scm.com - documentation officielle
- learngitbranching.js.org - tutoriel interactif
- github.com/github/gitignore - templates .gitignore
- choosealicense.com - choisir une licence
- Il existe beaucoup de tutoriels vidéo, guides et cours en ligne gratuits donc n'hésitez pas à chercher !

Créer son profil GitHub/GitLab ♣

À faire :

- Créer un compte avec un nom professionnel
- Ajoutez vos projets personnels et scolaires (publics)
- Mettez un lien vers votre git dans votre CV/LinkedIn
- Votre GitHub/GitLab vous sert de portfolio technique
- Ajoutez votre adresse mail ENSEA (settings -> emails)
- Activer votre compte étudiant (settings -> billing and licensing -> education benefits)

Vous pouvez déjà mettre votre TP de microcontroleur pour avoir un début de profil

BDTech

Instant promotionnel

Avec Athéna, nous allons commencer des projets ce semestre l'ocasion de mettre en pratique vos compétences en GIT. La présentation de ces projets se fera Lundi prochain, à la même heure.

Des TDs peuvent également être organisés si vous êtes intéressés.

Formulaire d'inscription et de feedback :

BDTech

Merci de votre attention !

Merci à Gauthier BIEHLER et Nicolas PAPAZOGLOU pour leur relecture et leurs suggestions.

Des questions ?

N'hésitez pas à me contacter pour toute question ou suggestion d'amélioration de cette formation.