[25-26] ISEN4 - Git [2H]

Sommaire

1. Introduction à Git et au Contrôle de Version

1.1. Introduction au contrôle de version

- Problèmes rencontrés sans système de versioning (perte de code, duplication).
- Notion de versioning centralisé vs décentralisé.

1.2. Git : c'est quoi ?

Git est un **système de gestion de version distribué**, conçu pour suivre les modifications apportées aux fichiers dans un projet, collaborer efficacement et gérer l'historique des changements.

1.2.1. Définition

Git permet de :

- **Suivre les modifications** apportées aux fichiers d'un projet, comme un code source.
- **Revenir à une version précédente** du projet en cas de besoin.

- **Travailler en équipe** sur le même projet, même si les collaborateurs sont dispersés géographiquement.

1.2.2. Origines

- Créé en 2005 par **Linus Torvalds**, le créateur de Linux.
- Développé pour gérer le code source de Linux avec des performances élevées et une grande fiabilité.

1.2.3. Caractéristiques principales

1. **Distribué** :
   - Contrairement à des outils centralisés comme SVN, chaque utilisateur dispose d'une copie complète du dépôt (l'ensemble des fichiers et leur historique).
   - Pas besoin d'une connexion constante à un serveur.
2. **Rapide** :
   - Les opérations (comme les commits, les fusions) sont rapides car elles sont effectuées localement.
3. **Historique sécurisé** :
   - Les données et l'historique sont stockés de manière immuable grâce à des mécanismes de hachage cryptographique (SHA-1).
   - Impossible de modifier l'historique sans que cela soit détecté.
4. **Branches flexibles** :
   - Les branches permettent de travailler sur des fonctionnalités ou corrections de bugs sans affecter la version principale.
   - Fusionner les branches est facile et efficace.

1.2.4. Pourquoi utiliser Git ?

1. **Travail collaboratif** :
   - Permet à plusieurs développeurs de travailler simultanément sur le même projet.
   - Les plateformes comme GitHub ou GitLab facilitent la gestion collaborative.
2. **Suivi des modifications** :
   - Historique complet des changements, avec les auteurs et les dates.
   - Identification et correction rapide des erreurs.
3. **Expérimentation** :
   - Créez des branches pour tester de nouvelles idées sans risquer de casser le projet principal.

1.3. Installation de Git

1.3.1. Téléchargement de Git

Sous Windows :

1. Aller sur le site officiel de Git : https://git-scm.com/.
2. Télécharger l'installeur pour Windows (version stable recommandée).
3. Lancer l'installeur et suivre les étapes :
   - **Choisir les options par défaut**.
   - Configurer l'éditeur par défaut (recommandé : Visual Studio Code ou Vim).
   - Sélectionner « Git from the command line and also from 3rd-party software ».
4. Terminer l'installation.

Sous Linux :

1. Ouvrir le Terminal.
2. Installer Git avec la commande adaptée à votre distribution :
   - **Debian/Ubuntu** :
     ```
     sudo apt update && sudo apt install git
     ```
   - **Fedora** :
     ```
     sudo dnf install git
     ```
   - **Arch Linux** :
     ```
     sudo pacman -S git
     ```
3. Vérifier l'installation avec `git --version`.

1.3.2. Configuration initiale de Git

1. Configurer le nom d'utilisateur global :
   ```
   git config --global user.name "your Name"
   ```
2. Configurer l'adresse email associée :
   ```
   git config --global user.email "your.email@example.com"
   ```
3. Vérifier la configuration :
   ```
   git config --list
   ```

1.3.3. Tester l'installation

1. Créer un dossier pour le test :
   ```
   mkdir test-git && cd test-git
   ```
2. Créer un fichier vide :
   ```
   echo "Hello world :)" > README.md
   ```

2. Concepts fondamentaux

2.1. Git comme base de données orientée contenu

2.1.1. Principe clé

Git ne stocke *pas* des fichiers, mais du **contenu identifié par son hash**.

C'est ce qu'on appelle un **CAS — Content Addressable Storage**.

2.1.2. Fonctionnement conceptuel

- Chaque élément stocké dans Git (un fichier, un répertoire, un commit…) est compressé, structuré, et **haché**.

- Le **hash** (anciennement SHA-1, désormais souvent SHA-256) détermine **l'adresse de l'objet**.
- Deux contenus strictement identiques → **même hash** → un seul stockage.

2.1.3. Conséquences conceptuelles majeures

- **Déduplication automatique** : Git ne stocke jamais deux fois la même information.
- **Immuabilité** : si le contenu change, le hash change → Git stocke un *nouvel* objet.
- **Référencement sûr** : l'adresse *est* le contenu → aucune ambiguïté possible.
- **Garantie d'intégrité** : si un bit change, le hash change → corruption détectable.

2.2. La philosophie "append-only"

Git a été conçu pour être **résistant aux corruptions**, **robuste** et **simple à répliquer**.

Pour cela, il adopte une stratégie d'écriture **uniquement additive** (*append-only*).

2.2.1. Propriétés du modèle append-only

- Git **n'écrase jamais** un objet existant.
- Git **n'efface rien** immédiatement.
- Toute modification crée **un nouvel objet** qui pointe vers les précédents.

2.2.2. Bénéfices de ce choix stratégique

- **Robustesse** : un commit ne change jamais → historique stable.
- **Traçabilité** : tout est traçable, rien n'est perdu.
- **Distribution** : très facile à copier/synchroniser (les objets sont immuables).
- **Sécurité** : le graphe d'historique devient une structure fiable et vérifiable.

Le revers :

- Certaines opérations (ex. nettoyage, rewriting) sont **complexes** car il faut recréer beaucoup d'objets.

2.3. L'espace conceptuel de Git : Worktree / Index / Repository

Git maintient **trois états conceptuels distincts**, tous indépendants du mécanisme des commandes.

2.3.1. **Worktree (Working Directory)**

C'est simplement le **répertoire de travail**, c'est-à-dire les fichiers *tels qu'ils existent physiquement* sur le disque.

→ Ce n'est **pas** Git : ce sont vos fichiers normaux.

2.3.2. **Index (Staging Area)**

C'est une **structure binaire interne**, optimisée, qui contient :

- une *photo* momentanée du contenu qui ira dans le *prochain commit*,
- des métadonnées (modification, mode, hash),
- un arbre partiellement construit.

Rôle conceptuel : l'index est un *buffer de snapshot*, pas un historique.

Pourquoi Git a besoin de l'Index ?

- Permet de définir ce qui *constitue* le prochain snapshot.
- Optimise massivement la performance (pas besoin de rescanner tout le repo).
- Découple le travail réel du travail versionné.

2.3.3. **Repository (.git/ directory)**

C'est la **base de données Git** elle-même :

- objets (blobs, arbres, commits, tags),
- refs,
- configuration,
- index,
- packfiles.

2.4. Le DAG : Directed Acyclic Graph

Git modélise l'historique sous forme de **graphe orienté acyclique**, pas comme une suite de versions.

2.4.1. Composition du DAG

- Un **commit** pointe vers **un ou plusieurs parents**.
  - 1 parent : commit normal
  - 2+ parents : merge commit
- Les commits sont des **noeuds immuables**.
- Les liens parentaux imposent **l'absence de cycle**.

2.4.2. Pourquoi un DAG ?

- Représente efficacement l'évolution parallèle (branches).
- Permet les merges naturels grâce au **Lowest Common Ancestor** (LCA).
- Rend l'historique **non linéaire et fidèle** à la réalité collaborative.
- Supporte les workflows complexes, rebase, cherry-pick, etc.

2.4.3. Conséquences clés

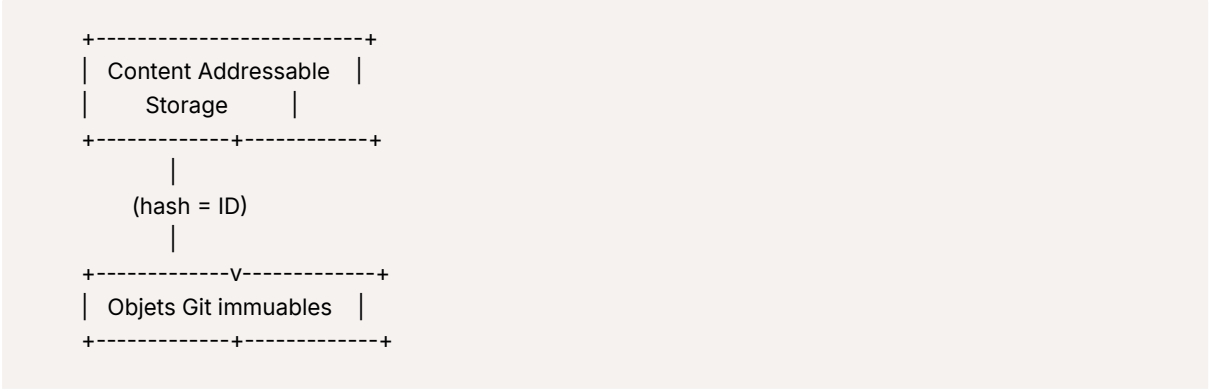
- Git n'impose pas une ligne du temps unique.
- L'ordre chronologique réel peut différer de l'ordre dans le DAG.
- L'unité sémantique est le « commit », pas la « version ».

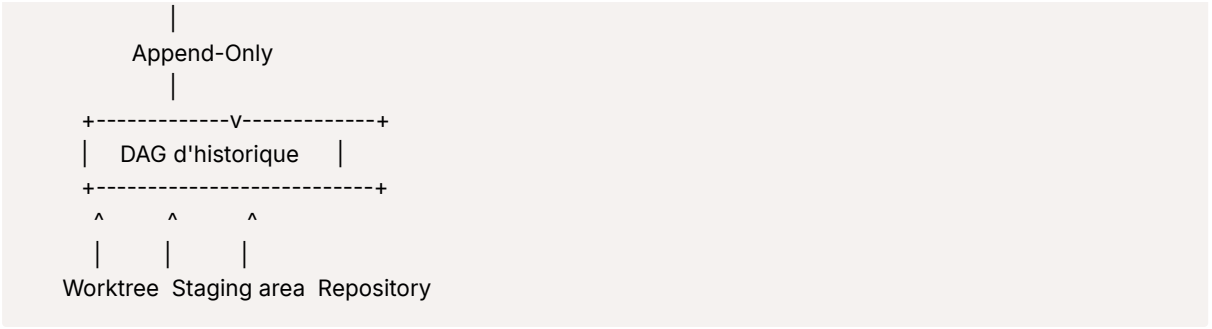
2.5. Bilan

2.5.1. Les trois zones de Git

Git fonctionne en trois zones principales :

1. **Working Directory** : Répertoire de travail contenant les fichiers.
2. **Staging Area** : Zone intermédiaire où on prépare les modifications pour le commit.
3. **Commit History** : Historique des modifications validées.





2.5.2. Cycle de base d'un fichier :

1. Modifier un fichier dans le répertoire de travail.
2. Ajouter ce fichier à la **staging area** avec `git add`.
3. Valider les modifications dans l'historique avec `git commit`.

3. Les bases de Git

3.1. Créer un dépôt Git local

Un dépôt Git est un répertoire dans lequel Git peut suivre les modifications des fichiers.

Commande pour créer un dépôt :

```
git init
```

- Cette commande initialise un dépôt Git dans le répertoire courant.
- Un dossier caché `.git` est créé. Ce dossier contient toutes les informations nécessaires pour que Git suive les fichiers.

Exemple :

```
mkdir my_project
cd my_project
git init
```

3.2. Cloner un dépôt distant

Si un projet existe déjà sur une plateforme distante comme GitHub, GitLab ou Bitbucket, il est possible de le cloner pour obtenir une copie locale.

Commande pour cloner un dépôt :

```
git clone URL_REPOSITORY
```

- L'URL peut être HTTPS, SSH, ou un chemin local.
- Le dépôt distant est copié dans un répertoire local, et Git configure automatiquement un remote (`origin`) pour le dépôt cloné.

Exemple :

```
git clone https://github.com/user/project.git
```

3.3. Les commandes de base

3.3.1. Vérifier l'état des fichiers

```
git status
```

- Affiche les fichiers modifiés, ajoutés ou supprimés.
- Indique les fichiers suivis/non suivis et ceux prêts pour un commit.

Exemple de sortie :

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
        modified: file.txt
```

3.3.2. Ajouter des fichiers à la staging area

```
git add file
```

- Prépare un fichier pour le commit.
- Pour ajouter tous les fichiers modifiés :

  ```
  git add .
  ```

3.3.3. Faire un commit

```
git commit -m "commit message"
```

- Un commit enregistre les modifications de la staging area dans l'historique.
- Le message doit être descriptif pour expliquer les changements.

3.3.4. Afficher l'historique des commits

```
git log
```

- Affiche la liste des commits avec leur hash, l'auteur, la date et le message.

Options utiles :

- Format simplifié :

  ```
  git log --oneline
  ```

- Ajouter un graphe visuel pour voir les branches :

  ```
  git log --graph --oneline
  ```

3.3.5. Exemple d'un workflow simple

1. **Créer un fichier dans le répertoire local :**

   ```
   echo "Hello Git :)" > file.txt
   ```

2. **Vérifier l'état des fichiers :**

```
git status
```

Sortie :

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

3. **Ajouter le fichier à la staging area :**

```
git add fichier.txt
```

4. **Vérifier de nouveau l'état :**

```
git status
```

Sortie :

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
     new file:   fichier.txt
```

5. **Faire un commit :**

```
git commit -m "Adding test file"
```

6. **Afficher l'historique :**

```
git log --oneline
```

3.4. Suppléments pour les bases

3.4.1. Annuler des modifications non ajoutées

Pour restaurer un fichier à son état initial (avant modification) :

```
git restore fichier
```

3.4.2. Retirer un fichier de la staging area

Pour retirer un fichier de la staging area sans perdre les modifications :

```
git restore --staged fichier
```

4. Architecture interne & Modèle de données

Cette section dévoile **la "machinerie interne" de Git**, ce qu'il fait réellement derrière les commandes.

C'est le cœur du fonctionnement : **objets, fichiers internes, hashing, indexation, compression et référentiels**.

4.1. Anatomie du répertoire `.git/`

Le dossier `.git/` est **le vrai dépôt Git**.

Le reste du répertoire n'est *que* le répertoire de travail.

Voici les principales composantes internes :

4.1.1. `objects/` — La base de données interne

Contient *tous* les objets Git :

- blobs (contenu de fichiers),
- trees (répertoires),
- commits,
- tags annotés.

Chaque objet est stocké dans un fichier dont le nom **est le hash** de son contenu → stockage *content-addressable*.

Organisation :

`objects/ab/cdef1234...`

- Les **2 premiers hex digits** = répertoire
- Le reste = fichier

Ce découpage évite d'avoir des milliers de fichiers dans un même dossier.

4.1.2. `refs/` — les pointeurs textuels

Les références sont de **simples fichiers texte** contenant un hash :

- `refs/heads/` → branches locales
- `refs/tags/` → tags
- `refs/remotes/` → branches distantes

Une branche est juste **un fichier avec 40 caractères**.

4.1.3. `HEAD` — Référence symbolique

`HEAD` pointe vers...

- soit un hash (mode détaché),
- soit une référence, ex. :

```
ref: refs/heads/main
```

HEAD **ne pointe jamais directement vers un commit**, mais vers la branche active.

4.1.4. `index` — Structure binaire optimisée

L'index (ou *staging area*) n'est **pas un fichier texte**.

C'est une structure binaire complexe contenant :

- le chemin de chaque fichier suivi
- son hash (blob)
- des métadonnées : taille, permissions, timestamps
- une structure d'arbre partiellement pré-assemblée
- un mécanisme d'accélération du diff (cache stat)

L'index est conçu pour la **performance** :

Git peut détecter des modifications sans rescanner tout l'arbre.

4.1.5. `packed-refs`

Quand beaucoup de branches/tags sont créés, Git regroupe les références dans un fichier textuel unique pour éviter la prolifération de petits fichiers.

4.1.6. `pack/` — Objets compressés pour transport et efficacité

Le répertoire pack contient des fichiers :

- `.pack`
- `.idx`

Git regroupe les objets dans un format ultra optimisé :

compression zlib + delta interne + trieur topologique → **réduction massive de taille**.

Les packfiles sont **la clé** de l'efficacité de Git pour les gros dépôts.

4.2. Les 4 types d'objets Git (le modèle de données)

Git n'utilise fondamentalement que **quatre types** d'objets, tous immuables.

4.2.1. Blob — le contenu brut

Un **blob** est l'équivalent d'un "fichier sans nom".

Pas de permissions, pas de date, pas de métadonnées.

Structure interne :

```
blob <size>\0<file-content>
```

→ Le hash est calculé sur cette représentation normalisée.

Caractéristiques :

- Git ne stocke le contenu qu'une seule fois.
- Deux fichiers identiques → *un seul blob*.

4.2.2. Tree — représentation d'un répertoire

Un **tree** contient :

- la liste des entrées enfants (blobs ou trees)
- leurs modes Unix (100644, 100755, 040000…)
- leur nom
- leur hash

Exemple conceptuel :

```
mode 100644 blob a1b2c3  file.txt
mode 040000 tree d4e5f6  src
```

Points clés :

- Les répertoires *n'existent pas physiquement* dans Git.
- Un tree est un *instantané* complet du répertoire.
- Changer un seul fichier → nouveau blob → nouveau tree → nouveau commit.

4.2.3. Commit — le nœud du graphe

Un commit contient :

1. **Le hash du tree racine**

2. **Le(s) parent(s)**
3. **L'auteur** (celui qui a écrit)
4. **Le committer** (celui qui a intégré)
5. **Le message**

Structure interne conceptuelle :

```
commit <size>\0
tree <hash>
parent <hash>
parent <hash>
author …
committer …
<message>
```

Le commit est **la seule entité qui forme le DAG**.

4.2.4. Tag annoté — un objet avec métadonnées

Il contient :

- le hash de l'objet étiqueté
- un nom
- un auteur
- un message
- éventuellement une signature cryptographique GPG

Très utile pour :

- releases officielles
- certification du code

4.3. Le mécanisme de hashage : SHA-1 / SHA-256

Git calcule le hash d'un objet sur la base :

```
type + espace + taille + \0 + contenu
```

Ainsi, des objets différents avec le même contenu n'ont pas le même hash.

4.3.1. Pourquoi ?

- Empêcher les collisions "inter-types" (ex. un blob qui ressemble à un commit).
- Assurer l'intégrité totale du graphe.
- Garantir que la structure interne est authentique.

4.3.2. Transition SHA-1 → SHA-256

Depuis 2020, Git supporte un mode SHA-256 :

- améliore la sécurité
- modifie tous les formats d'objet
- nécessite une compatibilité descendante entre dépôts

4.4. Packfiles : la compression à grande échelle

Git regroupe ses objets dans un format optimisé :

`.pack`

- contient tous les objets compressés
- peut stocker des deltas internes (diff binaires)

`.idx`

- index permettant de retrouver rapidement un objet dans le `.pack`

4.4.1. Pourquoi packer ?

- réduire la taille
- optimiser la réplication
- accélérer le transport réseau
- diminuer le nombre de fichiers (sinon millions)

4.4.2. Avantage clé :

Git peut *inventer* des deltas même si l'objet original n'a pas été stocké comme un diff.

4.5. Refs & HEAD : système de pointage minimaliste

4.5.1. Branche → fichier texte dans `refs/heads/`

Contenu : un hash.

4.5.2. Tag lightweight → pareil

Contenu : un hash.

4.5.3. Tag annoté → objet tag dans `objects/`.

HEAD → pointeur symbolique

Ex. `ref: refs/heads/main`

Ces pointeurs permettent d'attacher des *noms humains* à des *hashs immuables*.

4.6. Synthèse du Point 3

Git est construit comme :

- **une base de données immuable orientée contenu**
- un ensemble d'objets typés (blob, tree, commit, tag)
- un graphe d'historique basé sur les liens parentaux
- des références symboliques pour naviguer dans ce graphe
- des packfiles pour optimiser le stockage et le transport

Cette architecture explique :

- les performances extrêmes de Git
- sa robustesse
- sa facilité de distribution
- sa capacité à gérer des workflows complexes

5. Synchronisation, transport et distribution — Version approfondie

Git est un système **entièrement distribué**, ce qui implique que *chaque* dépôt contient l'intégralité de l'historique.

La synchronisation entre dépôts repose sur des mécanismes intelligents de **négociation d'objets**, de **packfiles**, et d'un **protocole de transport optimisé**.

5.1. Git est un système distribué : implications fondamentales

5.1.1. Git *n'a pas* de « serveur central »

Ce que GitHub ou GitLab appellent un "remote" n'est qu'un **autre dépôt complet**.

Chaque dépôt local contient :

- 100 % de l'historique
- tous les objets (blobs, trees, commits, tags)
- toutes les références
- le DAG entier

Ainsi, **toutes les opérations qui lisent l'historique sont locales**.

5.1.2. Conséquences majeures de cette architecture distribuée

1 — **Performance locale extrême**

Toutes les opérations complexes (log, diff, merge, blame…) se font **sans réseau**.

2 — **Résilience totale**

Pas de point de défaillance unique :

→ un remote peut disparaître, et personne ne perd l'historique.

3 — **Flexible : n'importe quel dépôt peut devenir un serveur**

Pas de hiérarchie, pas de "maître" infra.

4 — **Facile à bifurquer, cloner, répliquer**

Cloner un dépôt = obtenir une base de données complète.

5 — **Indépendance des workflows**

Git ne force pas de workflow particulier :

- centralisé simulé,
- décentralisé complet,
- pair-à-pair.

5.2. Le processus de synchronisation : "have" / "want" negotiation

Lors d'un échange entre deux dépôts (fetch, push, clone…), Git ne transfère **que les objets manquants**.

Ce mécanisme repose sur une négociation entre les deux dépôts.

5.2.1. Processus conceptuel résumé

Étape 1 — Le client envoie une liste de références qu'il souhaite (want)

Ex. :

- `refs/heads/main`
- `refs/tags/v1.0`

Étape 2 — Le serveur répond en envoyant des commits qu'il a (have)

Il parcourt son DAG et envoie :

- les commits
- puis leurs parents
- jusqu'à trouver un commit commun avec le client

Étape 3 — Détection du "common ancestor"

Le point où les deux DAG se rejoignent.

Étape 4 — Génération d'un packfile minimal

Le serveur génère un pack contenant **uniquement les objets nécessaires**, optimisés via :

- compression delta
- regroupement topologique
- tri des objets

Étape 5 — Envoi du pack au client

Le client l'intègre dans son `objects/pack/`.

5.2.2. Pourquoi cette négociation est indispensable ?

- Évite l'envoi d'objets déjà présents → ***efficacité massive***.
- Permet de transférer même des repositories **énormes** (millions d'objets).
- Supporte des flux avec faible bande passante (Git a été conçu pour gérer le kernel Linux).

5.3. Gestion du temps : causalité vs chronologie

Git ne se base **pas** sur le temps réel pour structurer l'historique.

5.3.1. Fondamentaux :

- Chaque commit pointe vers son ou ses parents
- Cette relation impose **l'ordre causal**
- Les timestamps humains (auteur / committer) sont secondaires

5.3.2. Conséquences :

1 — Le DAG n'est pas chronologique

Deux commits créés à des moments différents peuvent apparaître dans n'importe quel ordre dans l'historique global.

2 — Le merge restaure la causalité, pas la temporalité

La fusion préserve le graphe, pas le temps.

3 — Deux machines peuvent avoir des horloges non synchronisées

Git s'en moque tant que les parents sont cohérents.

4 — Rebase modifie le DAG

Changer les parents → changer l'ordre causal.

5.4. Identité : auteur, committer, signatures

5.4.1. Auteur vs committer

Git distingue deux identités :

👤 **auteur**

Celui qui *a écrit la modification*.

Correspond souvent à l'intention humaine.

👤 **committer**

Celui qui *a enregistré la modification dans le DAG*.

Peut être :

- une autre personne
- un script
- un outil d'intégration continue (CI)

Cette distinction existe pour des raisons de traçabilité.

5.4.2. Signatures cryptographiques (GPG / SSH)

Git permet de signer :

- des commits
- des tags
- des pushes

Objectif : garantir :

- l'authenticité
- l'intégrité
- la provenance
- la résistance aux attaques (supply-chain, falsification, etc.)

Une signature cryptographique protège le graphe :

→ un commit falsifié devient immédiatement détectable (hash incohérent).

5.5. Les protocoles de transport Git

Git peut transporter ses objets via plusieurs protocoles :

5.5.1. **Git Protocol natif (`git://`)**

- Très rapide
- Optimisé pour packfiles

  – Aucune sécurité (plaintext)

5.5.2. **SSH**

- Sécurisé
- Authentification forte
- Très utilisé en entreprise

  – Nécessite clés/gestion d'accès

5.5.3. **HTTPS**

- Très compatible
- Authentification fine (tokens OAuth2)

  – Un peu moins performant que SSH

5.5.4. **Local** (file://)

- instantané
- aucune sérialisation

– limité à la même machine

Tous reposent sur la même négociation interne.

5.6. Clone, Fetch, Pull, Push

5.6.1. Clone :

- construit un dossier local complet
- récupère tous les objets sous forme d'un pack
- installe les refs distantes comme `origin/main`

Si un projet existe déjà sur une plateforme distante comme GitHub, GitLab ou Bitbucket, il est possible de le cloner pour obtenir une copie locale.

Commande pour cloner un dépôt :

```
git clone URL_REPOSITORY
```

- L'URL peut être HTTPS, SSH, ou un chemin local.
- Le dépôt distant est copié dans un répertoire local, et Git configure automatiquement un remote (`origin`) pour le dépôt cloné.

Exemple :

```
git clone https://github.com/user/project.git
```

5.6.2. Fetch :

- négocie pour obtenir uniquement les objets *nouveaux* du remote
- met à jour `refs/remotes/...`
- ne modifie pas le DAG local

Pour télécharger les mises à jour du dépôt distant sans les intégrer immédiatement dans la branche :

```
git fetch
```

- Cette commande met à jour les références locales pour les branches distantes.

5.6.3. Pull :

Fetch + fusion ou rebase automatisé

Pour synchroniser une branche locale avec les dernières modifications du dépôt distant :

```
git pull
```

> *N.B : Cela combine git fetch et git merge.*

Exemple :

```
git pull origin main
```

5.6.4. Push :

- envoie uniquement les objets absents du remote
- refuse si cela briserait l'histoire distante (fast-forward protection)

5.6.5. Qu'est-ce qu'une pull request ?

Une pull request (PR) est une demande pour fusionner les modifications d'une branche dans une autre (souvent sur une plateforme comme GitHub ou GitLab).

Étapes pour créer une pull request :

1. **Pousser la branche** sur le dépôt distant :

   ```
   git push origin branch_name
   ```

2. Sur la plateforme distante, ouvrir une PR.
3. **Demander une revue** pour obtenir des commentaires de coéquipiers.
4. Une fois approuvée, fusionner la PR.

5.7. Synthèse du Point 4

5.7.1. Git repose sur trois idées fondamentales :

1. **Distribution totale**

   Chaque clone est un dépôt complet → performance, résilience, flexibilité.

2. **Négociation intelligente**

   Git ne transfère jamais trop : il construit des échanges minimaux et optimisés via des packfiles.

3. **Causalité > chronologie**

   Le DAG structure l'histoire par *lien parental*, pas par timestamp.

5.7.2. Résultat :

Git est un système de versionnement extrêmement performant, fiable, sécurisé et capable de supporter des projets gigantesques et massivement parallèles (ex. Linux).

6. Branches, merges et workflows

→ Qu'est-ce qu'une branche ?

Une branche est une version parallèle d'un projet.

- Par défaut, Git crée une branche appelée `main` ou `master`.
- Il est possible de créer des branches pour développer de nouvelles fonctionnalités ou corriger des bugs sans affecter le code principal.
- Une branche est simplement un pointeur vers un commit spécifique dans l'historique.

Exemple visuel simplifié :

```
main: A---B---C
feature:      \---D---E
```

- `main` continue sur son chemin principal (commits A, B, C).
- `feature` diverge pour travailler sur une fonctionnalité spécifique (commits D, E).

6.1. La branche Git : un simple pointeur mobile

Contrairement aux VCS traditionnels (CVS, Subversion), une branche Git n'est **pas** une copie du projet.

Elle est :

👉 **un fichier texte contenant un hash**.

Exemple conceptuel dans `refs/heads/main` :

```
1f4ac3b59ef7bd913a...
```

6.1.1. Implication fondamentale

Créer une branche n'a aucun coût :

- pas de duplication,
- pas de copie,
- pas de surcharge mémoire.

Git ne manipule que des **références**.

6.1.2. Bénéfices

- branches très légères → workflows extrêmement flexibles
- possibilité de créer des dizaines/milliers de branches
- navigation instantanée dans le DAG

6.2. HEAD et le mécanisme de "déplacement"

HEAD pointe vers la branche *active* :

```
ref: refs/heads/dev
```

Quand un nouveau commit est créé, Git :

1. crée un **nouvel objet commit**,
2. met à jour la branche (donc la ref) pour pointer vers le nouveau hash,
3. HEAD n'a pas changé de cible, mais la **ref qu'il suit bouge** → "ref moves forward".

Le mouvement du pointeur constitue *l'évolution de l'historique*.

6.3. La fusion : concept du "three-way merge"

Pour fusionner deux branches (ou plus), Git s'appuie sur son DAG.

6.3.1. Idée clé

Git ne fusionne pas deux projets, mais **trois snapshots** :

1. **la version sur la branche A**
2. **la version sur la branche B**
3. **leur ancêtre commun le plus récent** (LCA — *Lowest Common Ancestor*)

C'est le fameux **three-way merge**.

6.3.2. Calcul de l'ancêtre commun (LCA)

Git parcourt les parents des deux commits jusqu'à trouver

→ le premier commit reachable depuis les deux côtés.

Cet algorithme s'appuie sur une structure de graphe.

6.3.3. Pourquoi c'est critique ?

- Permet une fusion correcte même si les branches ont divergé depuis longtemps.
- Évite de considérer tout changement comme un conflit.
- Confère à Git sa *robustesse historique*.

6.3.4. Types de merge

Fast-forward merge

Cas où la branche cible n'a pas divergé :

```
A → B → C ← (nouvelle branche)
```

La branche cible avance simplement vers le commit C.

→ Aucun commit de merge.

→ Mise à jour rapide du pointeur.

Merge commit (three-way merge ordinaire)

Lorsque les branches ont divergé :

Git crée un **nouveau commit E** ayant **deux parents**.

L'histoire diverge puis se rejoint : structure naturelle du DAG.

Octopus merge (multi-parents)

Cas où Git fusionne **plus de deux** branches d'un coup.

Utilisé en intégration massive (rare dans flux standard).

6.4. Concept du rebase : réécriture du DAG

Un rebase est une **réécriture de l'histoire**, pas une fusion.

6.4.1. Idée clé

Git ne déplace pas un commit.

Git **recrée** de nouveaux commits avec :

- les mêmes changements,
- un nouveau parent,
- donc de nouveaux hashes.

6.4.2. Effet sur le graphe

On prend une séquence linéaire :

```
A → B → C
```

Et on la réapplique ailleurs :

```
X → Y → (nouveaux commits B′, C′)
```

Le DAG change radicalement, mais les fichiers finaux sont identiques.

6.4.3. Pourquoi rebase existe-t-il ?

- clarifier une histoire devenue trop complexe

- éviter les merges inutiles
- rendre l'historique linéaire (utile dans certains workflows)

Risques

Dans un dépôt distribué, réécrire l'histoire publique peut créer des incohérences, car les références ne pointent plus vers des commits existants.

6.5. Cherry-pick : extraction sélective

Concept :

→ appliquer un commit particulier (ou un ensemble) ailleurs dans le DAG.

Ce n'est **pas** une fusion, ni un rebase, ni une copie :

Git **recrée** le commit avec un nouveau parent.

Utile pour :

- backport de correctifs,
- patchs isolés,
- gestion fine d'hotfix.

6.6. Modèles de workflows : implications du modèle interne

Git ne dicte aucun workflow, mais **favorise certains modèles** à cause de son architecture.

6.6.1. Git Flow (model orienté release)

- branches structurées : `develop`, `feature/*`, `release/*`, `hotfix/*`, `main`
- adapté aux forts cycles de release
- clarifie les rôles des branches
- très verbeux, beaucoup de merges

Git Flow tire parti du **coût nul des branches** et de la **richesse du DAG**.

6.6.2. GitHub Flow (trunk lightweight)

- une branche principale (`main`)
- chaque fonctionnalité → "feature branch" courte
- merge via Pull Request
- CI/CD intégré

Simple, mais nécessite une intégration continue robuste.

6.6.3. Trunk-Based Development (TBD)

- branche principale unique (trunk)
- branches ultra-courtes (heures, pas jours)
- feature flags, tests très poussés
- excellent pour déploiements continus

Git excelle ici grâce à sa capacité à fusionner fréquemment et proprement.

6.7. Pourquoi Git rend certains workflows impossibles ou inutiles ?

6.7.1. Les branches ne "coûtent" rien

→ Multi-branches massifs possibles

6.7.2. Le DAG rend l'histoire non linéaire

→ forcer une chronologie stricte est artificiel

6.7.3. L'immutabilité rend certains workflows dangereux

→ réécrire l'histoire publique peut rompre la cohérence distribuée

6.8. Synthèse du Point 5

Git s'appuie sur des abstractions simples mais puissantes :

- une **branche = un pointeur**, rien d'autre
- un **merge = fusion de trois snapshots**
- un **rebase = réécriture du DAG**
- un **cherry-pick = duplication contrôlée d'un commit**

Ces mécanismes internes permettent :

- une collaboration massive,
- une flexibilité extrême du workflow,
- des stratégies complexes d'intégration,
- une haute résilience des historiques.

7. Exemple concret : un projet typique d'ingénierie logicielle

7.1. Contexte

Un groupe d'étudiants développe une **API web + interface frontend**.

L'équipe est composée de :

- Alice (backend)
- Bob (frontend)
- Chloé (devops & intégration)

Le projet évolue en parallèle et nécessite coordination + expérimentation.

7.2. Étape 1 — Initialisation du dépôt

L'équipe place le code initial dans Git.

7.2.1. Ce que Git fait réellement

1. Crée des **blobs** (fichiers),
2. Crée un **tree** (structure du projet),
3. Crée un **commit** qui pointe vers ce tree,
4. La branche `main` pointe vers le commit,
5. `HEAD` pointe vers la branche `main`.

Premier snapshot → racine du DAG.

7.3. Étape 2 — Travail en parallèle (multi-branches)

Alice crée une branche pour refactorer le backend,

Bob crée une branche pour refaire l'UI,

Chloé prépare une branche pour la CI.

7.3.1. Ce que Git fait en interne

Git crée trois **nouveaux pointeurs**, par exemple :

- `refs/heads/backend-refacto`
- `refs/heads/new-ui`
- `refs/heads/ci-pipeline`

Aucun fichier n'est dupliqué.

Ces branches ne sont que des étiquettes pointant vers le commit initial.

7.3.2. Pourquoi c'est important

- Git permet **plusieurs histoires parallèles**,
- Le DAG commence à s'étendre en plusieurs chemins,
- Chaque étudiant peut travailler sans interférer.

7.4. Étape 3 — Commits individuels

7.4.1. Exemple : Alice modifie `server/routes.py`

Git :

1. stocke un **nouveau blob** représentant le fichier modifié,
2. génère un **nouveau tree** pour le dossier `server`,
3. régénère un tree racine,
4. crée un **nouveau commit** pointant vers cette structure,
5. la branche `backend-refacto` avance vers ce commit.

7.4.2. Ce qui change vraiment

➡️ Git n'enregistre pas un "diff",

➡️ mais **un nouvel état complet du répertoire via les arbres** (avec réutilisation des blobs identiques).

7.5. Étape 4 — Travail divergent

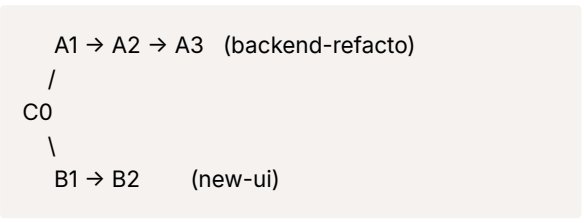
Pendant qu'Alice continue à modifier le backend,

Bob réorganise tout le CSS en frontend.

Le projet vit deux histoires parallèles :

- Historique A : backend
- Historique B : frontend

Le DAG ressemble à ceci :



Chaque chemin évolue indépendamment.

→ Git conserve l'intégrité de chaque branche grâce au graphe.

7.6. Étape 5 — Fusion du backend

Lorsque Alice a terminé, elle propose une fusion de `backend-refacto` dans `main`.

7.6.1. Ce que Git fait :

1. Cherche **l'ancêtre commun (LCA)** entre `main` et `backend-refacto`.
   Ici : `C0`.
2. Compare les snapshots de A3, C0 et M0 (le dernier commit de main).
3. Applique un **three-way merge**.
4. Produit un **nouveau commit de merge** qui a deux parents :
   - M0 (main)
   - A3 (backend-refacto)

Le DAG devient :

```
      A1 → A2 → A3
     /          \
C0 → M0 ------------ M1 (merge)
```

7.6.2. Intérêt

- Le DAG encode le fait que les deux histoires se rejoignent,
- La causalité est préservée,
- La structure reflète exactement comment les contributions ont divergé puis convergé.

7.7. Étape 6 — Rebase ou merge ?

Bob veut intégrer ses modifications UI.

Mais l'équipe préfère un **historique linéaire**.

Il utilise rebase *avant* fusion.

7.7.1. Ce que Git fait pendant un rebase :

Pour chaque commit B1 et B2 :

1. Git isole le *patch* induit par le commit,
2. Git crée un **nouveau commit** appliquant ce patch sur M1,
3. B1 devient B1′, B2 devient B2′,
4. Les anciens commits B1/B2 restent dans le DAG, mais leur branche ne les pointe plus.

Le graphe devient :

```
      A1 → A2 → A3
     /          \
C0 → M0 ------------ M1 → B1′ → B2′
```

7.7.2. Pourquoi rebase est utile ici ?

- L'historique devient lisible et linéaire,
- Les commits reflètent une progression causale cohérente.

7.8. Étape 7 — CI/CD (Chloé)

Chloé configure une pipeline (tests automatiques, déploiement).

Elle travaille dans une branche `ci-pipeline`.

Le merge de cette branche est simple, car elle partait de M0 et peu de fichiers se chevauchent.

→ Merge fast-forward possible, car `main` n'a pas divergé sur ces zones.

7.9. Étape 8 — Conclusion du projet

À la fin, le graphe représente **toute l'histoire causale** :

```
              B1' → B2'
             /
      A1 → A2 → A3
     /          \
C0 → M0 ------------ M1 -------- M2 (merge UI)
     \
      (fast-forward) → CI
```

Git encode :

- toutes les branches,
- tous les chemins d'évolution,
- toutes les convergences,
- l'intégralité des snapshots du projet.

7.10. Ce que cet exemple illustre

7.10.1. Git construit un graphe, pas une timeline

Chaque branche est une ligne parallèle du DAG.

7.10.2. Une branche = un pointeur, pas une copie

Créer une branche ne coûte rien → cela permet

➡️ de multiplier les fils d'exécution du projet.

7.10.3. Les merges reflètent la convergence logique

Le merge commit encode :

- deux parents
- un point de rencontre
- l'ancêtre commun utilisé pour fusionner

7.10.4. Le rebase est une réécriture du DAG

Les commits deviennent *d'autres commits*, liés différemment.

7.10.5. Le snapshot immuable rend Git robuste

Chaque version est un état complet du projet.

7.10.6. Le modèle distribué permet un travail parallèle massif

Chaque développeur peut travailler librement sans interférer.