
SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" Università di Roma
Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

# Programmazione WEB

*Autore*
Vincenzo Bova

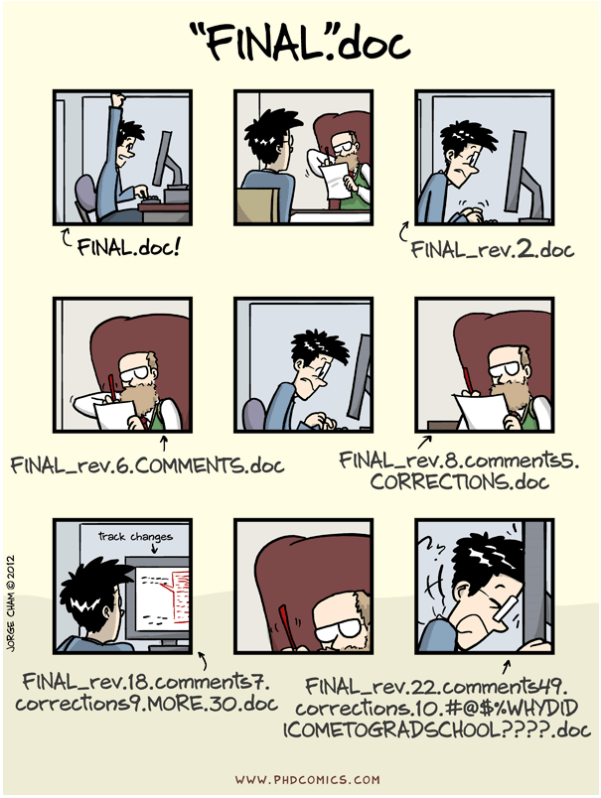
A.A. 2025/2026

# Indice

# 1 Introduzione a Git

## 1.1 Sistemi di versionamento

Durante lo sviluppo di un progetto c'è spesso la necessità di effettuare revisioni, correzioni o modifiche ai file che lo compongono.




Gestire ciò creando ogni volta nuovi file, tuttavia, comporta evidenti problemi:

- **Duplicazione del contenuto:** che rende il sistema inefficiente e aumenta la difficoltà nel mantenere integrità;
- **Assenza di Naming Convention:** che rende impossibile risalire ad uno storico delle modifiche;
- **Autori incerti**;
- ...

Per ovviare a ciò sono stati creati i **sistemi di versionamento** (git, csv, mercurial, svn...), i quali offrono vari benefici:

- **Gestione delle versioni:** il sistema si occupa automaticamente di etichettare le varie versioni in modo consistente;
- **Tracciamento delle mofiche:** è possibile accedere ad uno storico delle modifiche effettuate;
- **Presenza di metadati:** ogni modifica ha un autore, una data...;
- **Creazione di linee di sviluppo parallele:** è possibile creare una versione parallela del codice per non modificare la versione principale, e poi riunirle integrando i cambiamenti;
- **Sincronizzazione tra computer:** il sistema consente di mantenere il progetto allineato tra più computer.

## 1.2 Git

Git è un sistema di versionamento distribuito e veloce, creato nel 2005 e capace di gestire progetti di grandi dimensioni. Si basa su un design semplice e utilizza DAG (*Directed Acyclic Graph*) e Merkle trees come strutture dati.

**Definizione 1.1: Repository**

È un insieme di commit, branch e tag.
Per semplicità assumiamo che un progetto equivale ad un repository.

**Definizione 1.2: Working copy**

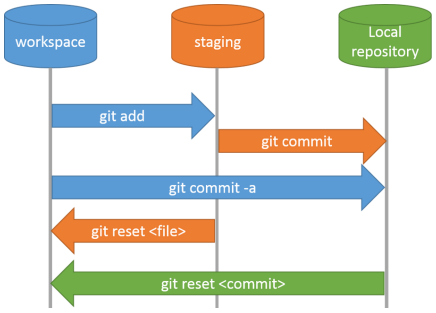
È l'insieme dei file tracciati nella copia locale del repository.
Quando creiamo un nuovo file non sarà ancora tracciato e bisognerà quindi aggiungerlo, quando invece modifichiamo un file già tracciato (*update*) stiamo aggiornando la working copy.

### 1.2.1 Commit

Un commit è un'istantanea del repository in un determinato momento.
Viene identificato dallo **SHA1** del commit stesso e contiene diversi campi:

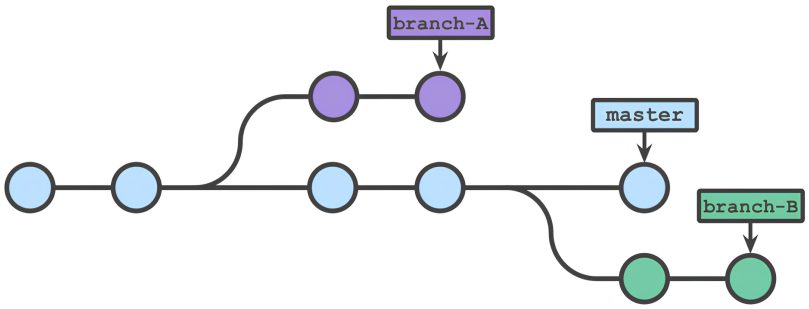
- data + autore, data + commiter
- commento **obbligatorio**
- 0,1 o più genitori
- tree: hash di tutti i file nel commit

In particolare il commit può contenere un sottoinsieme delle modifiche (anche ad un singolo file), le quali devono essere aggiunte alla staging area dei cambiamenti.

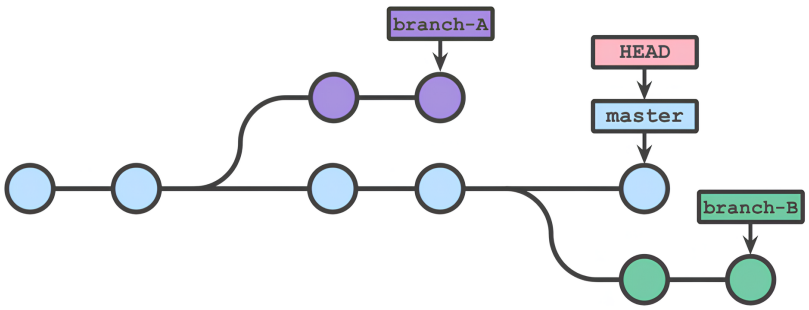

### 1.2.2 Branch

Un branch è una linea di sviluppo, composta da un insieme ordinato di commit collegati in un DAG, il quale inizia dal primo commit del repository e punta all'ultimo commit. Grazie ai branch è possibile **lavorare parallelamente** a più versioni del progetto.




### 1.2.3 HEAD

L'HEAD è un puntatore alla posizione attuale rispetto alla storia del repository e può essere aggiornato tramite il comando *checkout*.
Solitamente l'HEAD punta ad un branch o ad un tag, qualora invece puntasse ad un commit si parlerebbe di **Detached HEAD**. Quando ci si trova in questo stato i commit fatti non vengono inseriti in alcun branch, rischiando quindi di andare persi.




### 1.2.4 Tag

Un tag è un'etichetta per un commit e viene solitamente usato per segnare versioni importandi di un progetto (e.g. *v1.0.0*, *release-2025-09*).

### 1.2.5 Remotes

Sono dei riferimenti ai branch in repository remoti. Il nome predefinito è `origin` e vengono visualizzati nel formato `<remote>/<branch>` (es. `origin/main`).
In particolare definiamo come **tracking branch** un branch locale che tiene traccia di un branch remoto, facilitando l'uso dei comandi `git push` e `git pull`.

### 1.2.6 Merge

Il merge è un'operazione che fonde i cambiamenti apportati in due branch distinti, facendo in modo che il branch di destinazione contenga entrambi i cambiamenti e che quello di origine rimanga immutato.
Quando questa operazione viene eseguita tramite comando, Git determina in maniera autonoma quale tipo di merge sia più appropriato, basandosi sulla relazione tra i due branch e sullo storico dei loro commit.
In particolare esistono quattro tipologie di merge:

- **Fast forward:**
  - *Condizione:* il branch di origine è diretto discendente di HEAD.
  - *Azione:* Git sposta solo il puntatore di HEAD in avanti.
  - *Risultato:* nessun nuovo merge commit.
- **Merge commit:**
  - *Condizione:* i branch divergono e hanno sviluppi indipendenti;
  - *Azione:* Git combina le modifiche dei due branch, creando un nuovo commit;
  - *Risultato:* viene creato un merge commit con due genitori;
- **Rebase:**
  - *Condizione:* si vuole aggiorare un branch basandolo su un altro, riscrivendo lo storico;
  - *Azione:* Git ricrea ogni commit non in comune tra i due branch;
  - *Risultato:* la storia del branch diventa lineare, senza merge commit intermedi.
- **Three way:**
  - *Condizione:* storie divergenti (commit unici su entrambi i branch);
  - *Punti di confronto:*
    1. Base comune (Ancestor);
    2. Versione locale (HEAD);
    3. Versione remota (Branch);
  - *Azione:* Git crea un nuovo snapshot combinando le modifiche;
  - *Risultato:* Viene creato un merge commit.

## 1.3 Comandi Git

Per interfacciarsi con Git vengono messi a disposizione dal sistema diversi comandi:

- `git init`: inizializza un repository creando una subdirectory .git all'interno della directory corrente;
- `git status`: mostra lo stato attuale del repository (file tracciati, file modificati, file nello staging, file non tracciati);
- `git diff`: mostra le differenze tra working directory, staging e commit;
- `git add <file>`: aggiunge un file alla staging area (`git add .` per aggiungere tutti i file modificati);
- `git commit -m "Messaggio"`: crea un commit, registrando le modifiche aggiunte con `git add` nella cronologia del repository;
- `git log`: mostra la lista dei commit effettuati;
- `git branch <nome>`: crea un nuovo branch con il nome indicato, ma **non ci si sposta**;
- `git checkout <nome>`: passa ad un branch esistente spostando l'HEAD;
- `git checkout -b <nome>`: crea un nuovo branch con il nome indicato, per poi **spostarsi** su quest'ultimo (`git checkout -b <nome>` = `git branch <nome>` + `git checkout <nome>`);
- `git fetch`: scarica gli aggiornamenti (commit, branch) dal repository remoto, **senza merge** col tuo branch;
- `git merge <branch>`: unisce la cronologia del branch in cui ci si trova con quella del branch specificato;
- `git pull`: scarica gli aggiornamenti (commit, branch) dal repository remoto, **facendo merge** col tuo branch (`git pull` = `git fetch` + `git merge`);
- `git push`: invia i commit locali al repository remoto, aggiornando il branch remoto corrispondente;

## 1.4 Git flow

Con il termine Git flow intendiamo un modello di branching rigido per la gestione di rilasci e cicli di sviluppo definiti.
Il suo scopo è quello di separare gli ambienti di produzione, sviluppo, funzionalità e correzioni, attraverso i seguenti branch:

- `main/master`
  - **Contenuto:** solo codice stabile, testato e rilasciato.;
  - **Checkout da:** `release-*` o `hotfix-*`;
  - **Tag:** ogni merge riceve un tag di versione (es. *v1.0*);
- `develop`
  - **Contenuto:** cronologia completa delle funzionalità di sviluppo;
  - **Checkout da:** `feature-*`;
  - **Merge in:** `release-*`
- `feature-*`
  - **Scopo:** lavoro isolato su una nuova funzionalità;
  - **Checkout da:** `develop`;
  - **Merge in:** `develop`;
  - **Regola:** non interagisce mai con `main`;
- `release-*`
  - **Scopo:** preparazione per il prossimo rilascio;
  - **Checkout da:** `develop`;
  - **Attività:** Solo bug fixing minori e aggiornamento metadata (numero di versione);
  - **Doppio merge in:** `main` per il rilascio in produzione e `develop` per preservare le correzioni;
- `hotfix-*`
  - **Scopo:** Correzione immediata di bug critici trovati nel `main`;
  - **Checkout da:** `main`
  - **Doppio merge in:** `main` per deployare subito la correzione e `develop` per garantire che il bug non riappaia in futuro;

## 1.5 Gestione dei conflitti

Immaginiamo un contesto in cui tre sviluppatori lavorano allo stesso progetto:

- **Marco** (`fix-data-leakage`): si accorge di una falla critica nel preprocessing del dataset. Ha fatto 4 commit sul suo branch;
- **Luca** (`update-rules-parser`): ha aggiornato il parser delle regole della community, modificando gli **stessi file** di preprocessing toccati da Marco. Ha fatto 3 commit sul suo branch;
- **Voi** (`main`): effettuate il merge del lavoro di Marco senza problemi e ora dovete unire il lavoro di Luca.

Nel momento in cui proverete ad effettuare il secondo merge, Git non ve lo consentirà, mettendo in pausa il merge e marcando i file in conflitto nel seguente modo:

```
<<<<<<< HEAD
// Codice sul branch main
=======
// Codice sul branch update-rules-parser
>>>>>>> update-rules-parser
```

A questo punto sarà necessario risolvere i conflitti tramite l'interfaccia grafica aperta dal comando `git mergetool`, oppure manualmente aprendo ogni file, rimuovendo i marcatori di confitto e rieffettuando il commit.

## 1.6 Annullamento delle operazioni

- **Annulare in staging:** dopo aver eseguito `git add`, qualora non si volesse più committare il file, è possibile rimuoverlo dall'area di staging tramite il comando `git reset HEAD -- <file>`;
- **Annullare modifiche locali:** dopo aver modificato un file, è possibile scartare le modifiche e ripristinare quest'ultimo alla versione dell'ultimo commit tramite il comando `git checkout -- <file>`;
- **Annullare un commit pubblicato:** dopo aver eseguito un commit ed averlo pubblicato, è possibile annullarlo tramite `git revert <hash-commit>`;
- **Annullare un commit locale:** dopo aver eseguito un commit senza averlo ancora pubblicato, è possibile annullarlo tramite il comando `git reset <--soft|--hard> HEAD~1`, dove:
  - `--soft` rimuove l'ultimo commit mantenendo le modifiche nell'area di staging;
  - `--hard` rimuove l'ultimo commit cancellando completamente le modifiche;
  - `HEAD~1` indica il commit direttamente precedente ad HEAD (`HEAD~3` indica il 3°, etc...);
- **Riscrivere l'ultimo commit:** per rimpiazzare l'ultimo commit con uno nuovo è possibile utilizzare il comando `git commit --amend`.