

# INTRODUZIONE A GIT

Il sistema di versioning distribuito

# COS'È GIT?

- Git è un sistema di **controllo di versione distribuito** utilizzato per gestire il codice in progetti software.
-  A cosa serve?
-  Tiene traccia delle modifiche ai file nel tempo.
-  Permette a più sviluppatori di collaborare sullo stesso progetto.
-  BACK Consente di tornare a versioni precedenti del codice.
-  Come funziona?
- Repository locale: Ogni sviluppatore ha una copia completa del progetto.
- Commit: Salva una versione specifica del codice.
- Branch: Permette di lavorare su funzionalità diverse in parallelo.
- Merge: Unisce i cambiamenti di più sviluppatori.
-  Perché è utile?
-  Migliora la collaborazione
-  Previene la perdita di codice
-  Rende più semplice risolvere errori e bug



# PERCHÉ USARE GIT?



- Tracciamento delle modifiche



- Lavoro in team



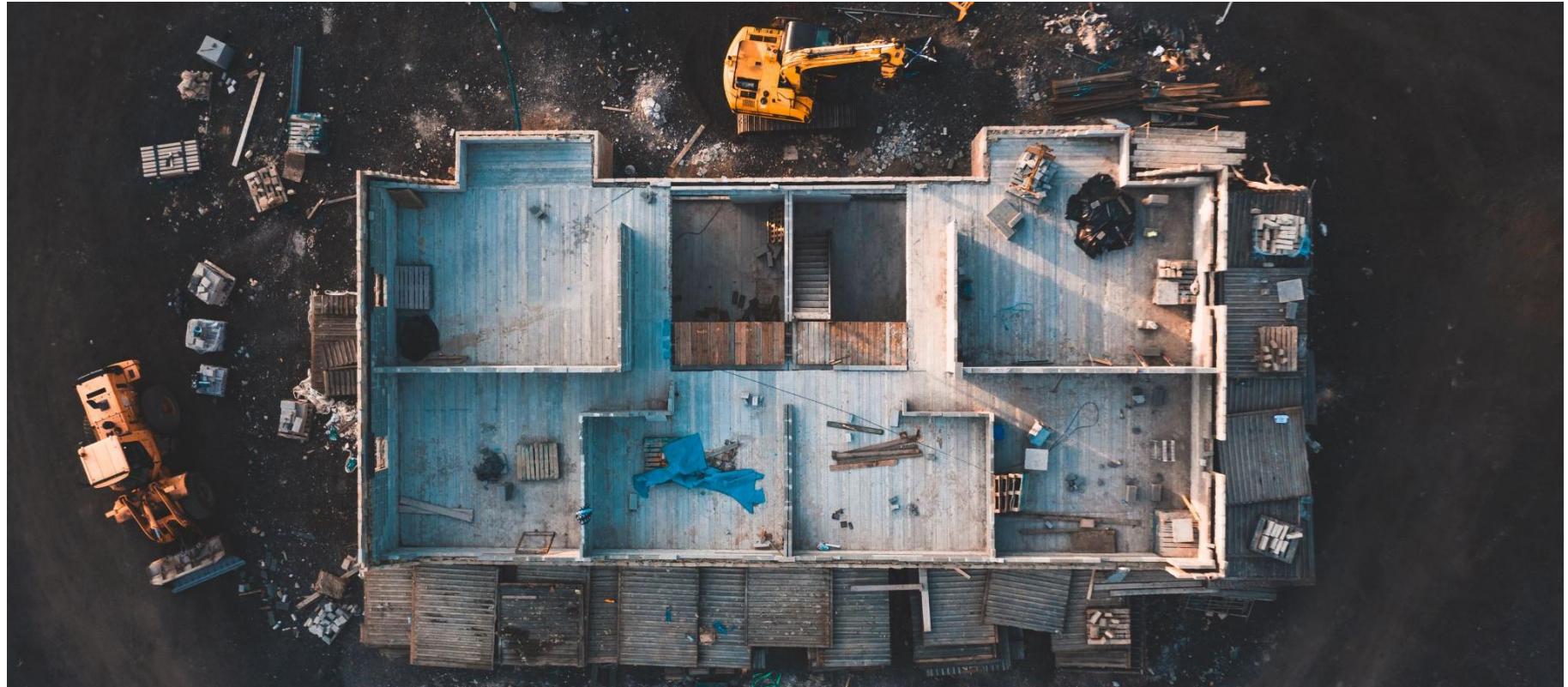
- Backup sicuro



- Sperimentazione senza rischi

# INSTALLAZIONE DI GIT

- Scarica Git da [git-scm.com](http://git-scm.com) e segui le istruzioni per l'installazione.



# CONFIGURAZIONE INIZIALE

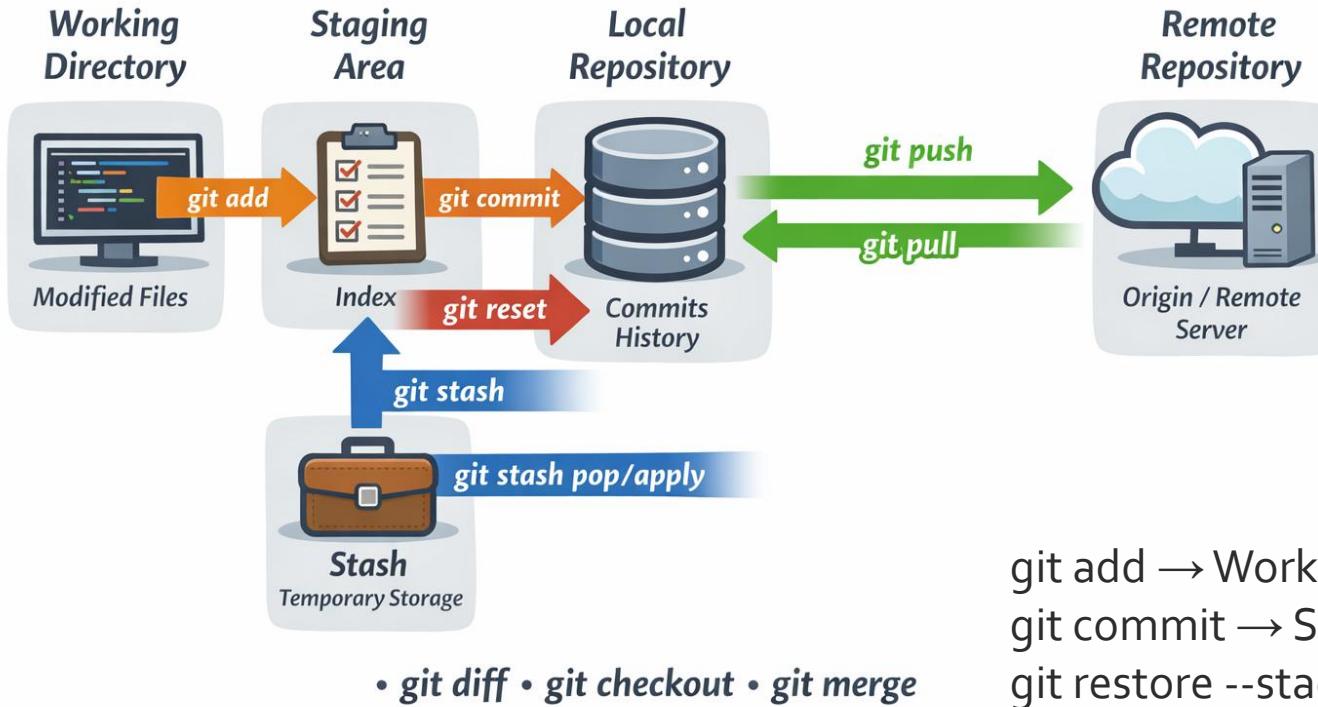


```
git config --global user.name "Il Tuo Nome"
```



```
git config --global user.email "tuo@email.com"
```

# GLI AMBIENTI



git add → Working → Stage  
git commit → Stage → Repo  
git restore --staged → Stage → Working  
git restore file → Repo → Working  
git reset --soft → Repo → Stage  
git reset --mixed → Repo → Working  
git reset --hard → ⚡ distruzione

Ogni comando Git è uno spostamento tra questi ambienti

# LE ISTRUZIONI PRINCIPALI

## 🔥 Istruzioni di base

Comando	Descrizione
<code>git init</code>	Inizializza un nuovo repository Git locale.
<code>git clone &lt;url&gt;</code>	Clona un repository remoto sul tuo computer.
<code>git status</code>	Mostra lo stato dei file (modificati, aggiunti o in staging).
<code>git add &lt;file&gt;</code>	Aggiunge un file all'area di staging.
<code>git commit -m "messaggio"</code>	Salva una versione del codice con un messaggio descrittivo.

## GIT INIT CREARE UN REPOSITORY GIT

- git init -> Inizializza un repository Git in una cartella

```
● PS C:\Corsi\git\repository-test> git init  
Initialized empty Git repository in C:/Corsi/git/repository-test/.git/
```

- git clone URL -> Clona un repository esistente

## GIT CLONE

- **git clone crea una copia completa di un repository remoto sul computer locale.**
-  Non copia solo i file, ma:
  - tutta la **storia dei commit**
  - i **branch**
  - il collegamento al **repository remoto**
- `git clone docentemaurocasadei/corso-git-2026`

# HEAD - MASTER

- **Cos'è HEAD**
- **È un puntatore speciale**
- Indica **dove ti trovi ora (quale branch)**
- Dice a Git:
  - “Questo è il commit corrente”
- **Cos'è master**
- **È un branch**
- Un branch **non è una cartella**
- È solo un **puntatore a un commit**
- HEAD → master → commit 16468b9
- Sto lavorando sul branch master e l'ultimo commit è 16468b9

- HEAD dice dove sei, il branch dice dove stai lavorando.  
Se HEAD non punta a un branch, sei in pericolo.

## ● Situazione normale



## ● Detached HEAD



## GIT ADD | AGGIUNGERE LE MODIFICHE ALLO STAGE

- In Git, il comando `git add .` (o `git add *`) serve per aggiungere le modifiche all'area di staging, ovvero un'area temporanea dove Git tiene traccia dei file che vuoi includere nel prossimo commit.

```
git add .
```

- Oppure
  - `> git stage nomefile`

# .GITIGNORE IGNORARE FILE

- **.gitignore è un file che dice a Git quali file o cartelle NON devono essere tracciati.**
-  Serve per **escludere file locali, temporanei o sensibili** dal repository.
-  **Perché usare .gitignore**
  - evitare file inutili (node\_modules/)
  - non caricare dati sensibili (.env)
  - tenere il repository **pulito**
  - evitare conflitti tra sviluppatori
  -
-  **Dove si trova**
  - nella **root del progetto**
  - può esistere anche in **sottocartelle**
  - vale per **tutti i file non ancora tracciati**

## Sintassi base

```
text  
  
# Commento  
nomefile.ext  
cartella/  
*.log
```

# GITIGNORE

- Caso in cui si aggiunge un file in .gitignore successivamente al tracciamento delle modifiche (stage)

Rimuovi il file dal versioning (NON dal disco)

```
git rm --cached nomefile
```

--cached = rimuove **solo da Git**, non dal filesystem

- **Crea una commit**
- `git commit -m "rimosso file riservato dal versioning"`

# GIT STATUS | STATO DEL REPOSITORY

- git status -> Mostra lo stato attuale dei file

```
● PS C:\Corsi\git\repository-test> git status
On branch dev2
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   prodotti.html

no changes added to commit (use "git add" and/or "git commit -a")
○ PS C:\Corsi\git\repository-test>
```

```
● PS C:\Corsi\git\git-test2> git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   read.me

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

# .GITIGNORE | SMETTERE DI TRACCIARE UN FILE

- Un file è già versionato e vuoi che Git smetta di tracciarlo.
- ⚠ .gitignore da solo **NON funziona** sui file già tracciati.
- Comandi:
  - git rm --cached secreto.txt
  - git commit -m "rimosso file dalla versione, ora ignorato"
- **Risultato**
  - ✓ Il file resta sul computer
  - ✓ Git non lo traccia più
  - ✓ Le regole di .gitignore ora funzionano

# GIT COMMIT

- Porta i file dallo stage e lo mette nel repo locale; NON prende i file dalla working directory  
prende solo ciò che è nello stage
- `git commit -m "messaggio del commit"`
-  **Cosa succede con git commit**
- viene creato un **commit**
- viene aggiornato il **branch corrente**
- HEAD si sposta sul nuovo commit
- la working directory resta invariata

# GIT COMMIT –A (STAGE DEI FILE TRACCIATI)



- I file già tracciati sono quelli già presenti in un commit
- git commit -a -m 'messaggio'
- -a = aggiunge automaticamente allo stage tutte le modifiche
- ⚠ Non include i file nuovi (serve git add)
- Commit rapido senza passare dallo stage manuale

# STASH

- 👉 **git stash mette temporaneamente da parte le modifiche non committate**, permettendoti di tornare a una working directory pulita **senza perdere nulla**.
- # modifichi dei file
- git status
- # vedi modifiche non committate
- git stash

## 🧠 Riepilogo rapido (slide-ready)

Obiettivo	Comando	🔗
Tutti i file tracciati	<code>git stash</code>	
Solo alcuni file	<code>git stash push -p</code>	
Inclusi non tracciati	<code>git stash -u</code>	
Anche ignorati	<code>git stash -a</code>	

↓

# STASH | RIPRENDERE I FILE

- git stash list
- Riprendere le modifiche (e rimuovere lo stash)
- git stash pop
  - ✓ Le modifiche tornano nei file
  - ✓ Lo stash viene **eliminato**
- Riprendere le modifiche (**senza eliminare lo stash**)
- git stash apply
  - ✓ Le modifiche tornano nei file
  - ✓ Lo stash **resta nella lista**

# STASH | PIU STASH

- Vedere gli stash salvati
- git stash list
- Es:
  - stash@{0}: WIP on main: modifiche temporanee
  - stash@{1}: WIP on feature-login: test
- **Come leggerlo**
  - stash@{0} → stash **più recente**
  - stash@{1} → stash precedente
  - WIP on <branch> → branch di origine
  - descrizione → contesto delle modifiche
- Per rimettere i file dello stash o in stage:
  - git stash pop **"stash@{0}"**

# GIT SHOW | VEDERE UN FILE IN UN DETERMINATO COMMIT

- git show permette di visualizzare il contenuto o le modifiche di una commit passata, senza cambiare lo stato del repository.
- 🤝 È un comando **solo in lettura**.
- git show <commit>:<percorso/file>
  
- git show 09ae73e:esercizi/torestore.html
  
- Per capire hash del commit:
- git log --oneline
- E poi
- git show 09ae73e:esercizi/torestore.html

# ANNULLARE UNA COMMIT O RECUPERARE UN FILE?

- **1** una COMMIT?
  - ┌── NON pushata → git reset
  - └── pushata → git revert
- **2** un FILE?
  - └── qualsiasi caso → git restore (--source se serve)

# GIT RESET – RITORNARE AD UN COMMIT PRECEDENTE 8 ANCHE SUCCESSIVO)

- git reset serve per tornare indietro nel tempo e annullare modifiche in Git.
- Puoi usarlo per rimuovere commit, modifiche o file dall'area di staging, senza perdere definitivamente i tuoi file.
- **SOLO se non già scritto su repo remoto**
- git reset --soft HEAD~1 \_ torna indietro di 1 commit
- git reset --soft **8df4b26** torna a quel commit

🤓 Quando si usa `git reset` ?

Situazione	Quale usare
Ho committato troppo presto e voglio aggiungere altro	--soft
Ho committato per errore e voglio correggere il codice	--mixed
Voglio cancellare tutto e ricominciare da zero	--hard

## GIT RESET | --SOFT E --MIXED

- git reset --soft
  - Mantiene tutto nell'index (staging area).
  - Non tocca il working directory (i file modificati rimangono intatti).
  - Sposta semplicemente il puntatore di HEAD al commit specificato.
  - Le modifiche risultano ancora "staged", pronte per essere committate di nuovo.
  - Uso tipico: Quando vuoi cambiare l'ultimo commit senza perdere le modifiche già aggiunte all'index.
  - -----
- git reset --mixed (di default)
  - Resetta l'index (staging area), ma mantiene intatti i file nel working directory.
  - Le modifiche vengono "un-staged" (rimosse dall'index), ma i file modificati rimangono nel progetto.
  - Uso tipico: Quando vuoi annullare l'ultimo commit e fare delle modifiche ai file prima di aggiungerli di nuovo.

# GIT RESET | --HARD (DISTRUGGE LE MODIFICHE)



- Serve per annullare commit locali NON pushati
- --hard → **distrugge tutto** !
- **! Usare SOLO se il commit NON è sul remoto**
- git reset --hard HEAD~1
- Significa:
  - sposta HEAD indietro
  - sposta il branch
  - cancella i file nella working directory
  - elimina il commit dalla storia locale
-  Per Git è come se quel commit non fosse mai esistito.
- Ma se in remoto esisteva A B C e in locale cancello C e poi PUSH, gli altri colleghi del team si troveranno disallineati in locale...

- Nota:

```
bc4ca38 HEAD@{1}: commit: mess5
77d98e9 (HEAD -> main) HEAD@{2}: commit: mess4
```

- git reset --hard 77d98e9

- Ritorna a quella commit ripristinando i file

```
77d98e9 (HEAD -> main) HEAD@{0}: reset: moving to 77d98e9
bc4ca38 HEAD@{1}: commit: mess5
77d98e9 (HEAD -> main) HEAD@{2}: commit: mess4
```

- git reset --hard bc4ca38

```
PS C:\Corsi\git\git-test3> git reflog
bc4ca38 (HEAD -> main) HEAD@{0}: reset: moving to bc4
77d98e9 HEAD@{1}: reset: moving to 77d98e9
bc4ca38 (HEAD -> main) HEAD@{2}: commit: mess5
77d98e9 HEAD@{3}: commit: mess4
```

- È ritornato «avanti»

# GIT RESET | DETTAGLI

- Eliminare l'ultima commit (locale, non pushata)

- `git reset --soft HEAD~1`

```
git reset --soft HEAD~1
```

- eliminare completamente anche le modifiche dai file:

- `git reset --hard HEAD~1`

- Se hai già pushato e vuoi eliminarla dal repository remoto (**può causare problemi agli altri sviluppatori**):

- `git reset --hard HEAD~1`

- `git push origin --force`



- Cos'è git reflog
- git reflog registra **tutti gli spostamenti di HEAD**
- Tiene traccia delle azioni locali:
  - commit
  - checkout
  - reset
  - merge
- Funziona **anche se i commit non sono più visibili nei branch**
- A cosa serve
  - 🔥 Recuperare commit dopo:
    - git reset --hard
    - git checkout
    - cancellazione o spostamento di branch
  - 🕵️ Debug di operazioni Git “andate male”
- Le entry del reflog **scadono** (di default ~90 giorni)
- Git Reflog – Esempio pratico
- Sono su b8539b1 (ref 2, HEAD)
- git reset 7bb8055 → HEAD torna a ref 1, file restano di ref 2
- git reflog → mostra ancora b8539b1
- git reset b8539b1 --hard
- ✅ Repository ripristinato allo stato dell'ultima commit

# GIT RESTORE | RECUPERARE LA VERSIONE DI UN FILE

- git restore --staged funziona SOLO PRIMA del commit  
👉 NON può agire su commit già fatte

- `git restore --staged app.js`
  - → rimuove il file dallo stage
  - → NON annulla le modifiche nel file
  - → funziona solo prima del commit

- `git restore app.js`
  - → sostituisce il file nella working directory
  - → con la versione del repository (HEAD)
  - → annulla le modifiche locali

- Se non esiste ancora alcun commit:
  - → non esiste HEAD
  - → git restore file NON funziona
  - → usare `git restore --staged` o `git reset file`

# GIT RESTORE –SOURCE= | RITORNARE ALLA VERSIONE DI UNA COMMIT PRECEDENTE

- **git restore --source** permette di ripristinare un file com'era in una commit precedente, senza riscrivere la storia Git.
-  È il metodo **corretto e sicuro** quando la commit è già pushata.
- **git restore --source=<commit> <percorso/file>**
- Esempio
- **git restore --source=o9ae73e esercizi/torestore.html**

# GIT REVERT

- 👉 git revert annulla le modifiche di un commit creando un *nuovo commit*.  
👉 NON cancella né modifica la cronologia esistente, ma annulla le modifiche della commit e le rimette in working tree

## 📐 Esempio visivo

Prima

css

A — B — C — D    (**main**)

git revert C

mathematica

A — B — C — D — E    (**revert di C**)

👉 Il commit **C** esiste ancora, ma il suo effetto è annullato da **E**.

# GIT LOG - STORICO DELLE MODIFICHE

- git log -> Mostra la cronologia dei commit

```
PS C:\Corsi\ITS Turismo Marche\2024-2026 FullStack Senigallia\database-er> git log --oneline --graph --all --decorate
commit b7ffd3234265417b1c6b34115aa77a2e1f1f8719 (HEAD -> main)
Author: soluzione-software <info@soluzionesoftware.com>
Date:   Mon Mar 3 13:06:50 2025 +0100

    test

commit f82adaffac05b48eeb38b9ddcd2a4664ac0d8168 (origin/main)
Author: soluzione-software <info@soluzionesoftware.com>
Date:   Mon Mar 3 12:50:00 2025 +0100

    DISPENSA AGGIORNATA

commit 58936b41cad76d320b22ffa8a263851b7644ef83
Author: soluzione-software <info@soluzionesoftware.com>
Date:   Fri Feb 28 16:27:57 2025 +0100
```

- git log --oneline # versione compatta
- git log --graph # mostra il grafo dei branch
- git log --oneline --graph --all

```
PS C:\Corsi\git\git-test2> git log --oneline --graph --all
>>
* 16468b9 (HEAD -> master) secondo
* 35446d7 primo
* 5f35f9b primo
```

```
git log --oneline --graph --all --decorate
```

```
PS C:\Corsi\git\git-test3> git log --oneline --graph --all --decorate
* daa10dc (HEAD -> main) f1 a
* b4eb95b hh
| * cb864df (mod4) jj
|
* b52c04b (origin/main, origin/HEAD) Update file1.txt
* b606f5e hh
* a4477ee Update file3.txt
* -- 7cfc55a ...
```

```
PS C:\Corsi\git\git-test2> git log --oneline
16468b9 (HEAD -> master) secondo
35446d7 primo
5f35f9b primo
```

```
PS C:\Corsi\git\git-test2> git log --graph
* commit 16468b93a211448ab76578c190d610661fd9116e (HEAD -> master)
| Author: Mauro Casadei <info@soluzionesoftware.com>
| Date:   Thu Jan 8 22:03:40 2026 +0100
|
| secondo
|
* commit 35446d71eb745ad1a34c106e21cabcb9b166e7996
| Author: Mauro Casadei <info@soluzionesoftware.com>
| Date:   Thu Jan 8 22:03:02 2026 +0100
|
| primo
|
* commit 5f35f9b0df7226dcc175247bcea0fdee79f0becf
```

# GIT LOG | GESTIONE DELLA CRONOLOGIA

Comando	Descrizione
git log	Mostra la cronologia dei commit del branch corrente.
git log --oneline	Mostra la cronologia in forma compatta (1 riga per commit).
git log --graph --all --decorate	Visualizza la storia completa con struttura dei branch.
git reset --soft <commit>	Torna a un commit precedente mantenendo le modifiche nello <b>stage</b> .
git reset --mixed <commit>	Torna a un commit precedente mantenendo le modifiche nella <b>working directory (default)</b> .
git reset --hard <commit>	<span style="color: orange;">⚠️</span> Torna a un commit precedente <b>annullando tutte le modifiche</b> .
git revert <commit>	Crea un <b>nuovo commit</b> che annulla le modifiche di un commit precedente.
git reflog	Mostra la cronologia dei movimenti di HEAD (recupero di commit persi).

# GESTIONE DEI BRANCH

-  Gestione dei branch (versione aggiornata)
- `git branch <nome-branch>` Crea un nuovo branch.
- `git switch <branch>` Passa a un branch esistente.
- `git switch -c <branch>` Crea e passa a un nuovo branch.
- `git checkout <branch>` Passa a un altro branch (comando storico)
- `git merge <branch>` Unisce un branch con quello corrente.
- `git branch -d <branch>` Elimina un branch locale.
- `git branch -M main` Rinomina il branch corrente in main

## GIT BRANCH \*\*\*\* |CREARE E GESTIRE BRANCH

- git branch nomebranch -> Crea un nuovo branch
- Oppure git checkout -b nomebranch
- git checkout nomebranch -> Cambia branch

```
git checkout -b dev2
```

```
git branch dev  
git checkout dev
```

## GIT BRANCH | MOSTRARE I BRANCH

- Git branch (lista branch)
- Git branch –r (lista branch remoti)

```
● PS C:\Corsi\git\repository-test> git branch
  dev
* dev2
  master
```

# GIT SWITCH | MODIFICA IL BRANCH UTILIZZATO

- Passare a un branch esistente
  - git switch nome-branch
- Creare **e** passare a un nuovo branch
  - git switch -c nuovo-branch
- Oppure: git checkout
  - git checkout nome-branch
- Creare **e** passare a un nuovo branch
  - git checkout -b nuovo-branch

## Comando

## Cosa fa

git switch

**Solo** cambio branch

git checkout

Cambio branch **e** ripristino file/commit



# GIT BRANCH | CREARE UN BRANCH DA UNA COMMIT



- git branch mod9 adb2cac
- Il branch mod9 punta alla commit adb2cac
- Non parte dall'ultima commit (HEAD)
- git switch mod9 sposta HEAD su quel ramo
- Un branch è solo un puntatore a una commit

```
● PS C:\Corsi\git\git-test-2026> git log --oneline
76eecb5 (HEAD -> master) messaggio
b8539b1 ref 2
7bb8055 ref 1
adb2cac 9r ←
34db2af mod 5r
```

```
\ No newline at end of file
● PS C:\Corsi\git\git-test-2026> git branch mod9 adb2cac
● PS C:\Corsi\git\git-test-2026> git switch mod9
  Switched to branch 'mod9'
● PS C:\Corsi\git\git-test-2026>
```

ho creato un branch da un commit precedente

# DETACHED HEAD (ERRORE / PROBLEMA)

- È una situazione di pericolo, un commit potrebbe fare perdere le modifiche in quanto non si è in un branch
- Si verifica quando (git checkout su commit):
  - git log --oneline --graph --all
  - >>
  - \* 16468b9 (HEAD -> master) secondo
  - \* 35446d7 primo
  - \* 5f35f9b primo
- PS C:\Corsi\git\git-test2> git checkout 16468b9
  - Se si fa git commit in detached mode le modifiche verranno **garbage-collected** prima o poi
  - Il commit non è più rintracciabile

## GIT MERGE | UNIRE DUE BRANCH

- git merge nomebranch -> Unisce le modifiche di un branch in un altro

- PS C:\Corsi\git\repository-test> **git merge dev**  
Updating 5fdcd06..3681da5

## GIT BRANCH -D | ELIMINARE UN BRANCH

- Per eliminare un branch locale:
- `git branch -d nome-del-branch`
- Se il branch **non è stato unito** e vuoi forzare l'eliminazione:
- `git branch -D nome-del-branch`

```
● PS C:\Corsi\git\repository-test> git branch -d dev2  
Deleted branch dev2 (was f464da7).
```

# ORIGIN: COSA È?

- 👉 origin è il NOME DI UN REPOSITORY REMOTO

- I branch remoti hanno questa forma:

- origin/main
- origin/dev
- origin/m1

- 👉 Qui:
  - origin = repository remoto
  - main, dev, m1 = branch dentro quel repository

```
Repository remoto: origin
|
└── main      → origin/main
    ├── dev    → origin/dev
    └── m1     → origin/m1
```

# GESTIONE DEL REPOSITORY REMOTO

## **Gestione del repository remoto**

Comando	Descrizione
<code>git remote add origin &lt;url&gt;</code>	Collega il repository locale a quello remoto.
<code>git push origin &lt;branch&gt;</code>	Invia i cambiamenti al repository remoto.
<code>git pull origin &lt;branch&gt;</code>	Scarica e unisce le modifiche dal remoto.
<code>git fetch</code>	Recupera i dati dal remoto senza unirli.

# CREARE REPOSITORY REMOTO SU GITHUB + GIT REMOTE ADD

- echo "# repository-test" >> README.md
- git init
- git add README.md
- git commit -m "first commit"
- git branch -M main
- git remote add origin https://github.com/docentemaurocasadei/repository-test.git
- git push -u origin main
- ...or push an existing repository from the command line
- git remote add origin https://github.com/docentemaurocasadei/repository-test.git
- git branch -M main
- git push -u origin main
- → -u = upstream: traccia un riferimento fisso tra quei 2 branch

# GIT REMOTE ADD | COLLEGARE UN REPOSITORY GIT LOCALE A UN REPOSITORY REMOTO (PER ESEMPIO SU GITHUB, GITLAB, BITBUCKET).

```
PS C:\Corsi\git\repository-test> git remote add origin https://github.com/docentemamurocasadei/repository-test.git
>>
PS C:\Corsi\git\repository-test> git branch -M main
>>
PS C:\Corsi\git\repository-test> git push -u origin main
>>
Enumerating objects: 18, done.
Counting objects: 100% (18/18), done.
Delta compression using up to 12 threads
Compressing objects: 100% (17/17), done.
```

## GIT REMOTE -V

- Vedere quali repository remoti sono configurati
- git remote -v
- origin https://github.com/docentemaurocasadei/corso-git-2026 (fetch)
- origin https://github.com/docentemaurocasadei/corso-git-2026 (push)
  - nome del remoto (origin)
  - URL
  - se usato per fetch e/o push

# GIT REMOTE RENAME | RINOMINARE UN REPOSITORY REMOTO

- git remote rename origin upstream
- Rinomina un repository remoto già configurato
- git remote rename origin upstream
- ✓ Il remoto prima chiamato origin
  - ✓ ora si chiama upstream

# GIT REMOTE | DISCONNETTERE UN REPOSITORY REMOTO

- `git remote remove origin`
- **Quando serve**
- Il repository remoto non è più valido
- Hai collegato l'URL sbagliato
- Vuoi cambiare piattaforma (GitHub → GitLab)
- Lavori solo in locale

# GIT FETCH

- **git fetch — scarica ma NON modifica**
- **👉 Recupera le novità dal repository remoto, ma non tocca il tuo codice locale.**
- scarica nuovi commit dal remoto
- aggiorna i branch remoti (origin/main, ecc.)
- NON fa merge
- NON cambia file

# GIT PULL

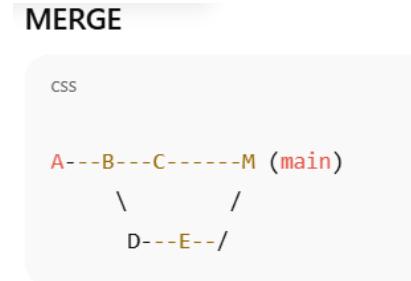
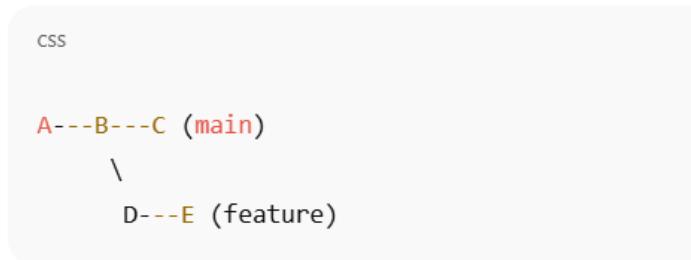
- È un fetch + merge (o rebase).
- ➡ Recupera le novità dal repository remoto, e unisce con il branch.
- >git pull equivale a:
  - git fetch
  - git merge origin/<nome branch>
- ✖ Modifica il tuo branch locale.

## Rischio

- può creare merge commit
- può causare conflitti
- può “sporcare” la storia

# GIT PULL --REBASE

- Rebase = riscrive la storia dei commit, invece che fare un merge applica le modifiche in fondo alla storia dei commit
- merge → incrocio
- rebase → linea dritta
- Con 2 branch:
  - Prima (branch divergenti)



I commit D ed E vengono ricreati sopra C.

- Significa: “Rifai i miei commit come se fossero partiti dall’ultimo main”.

git pull –rebase

Equivale a:

git fetch

git rebase origin/main

**GIT PUSH | INVIA LE MODIFICHE LOCALI AL REMOTO**

- **git push** invia i commit dal repository locale al repository remoto.
  -  Rende le modifiche **visibili e condivise** con gli altri.



## GIT LOG | REPO REMOTO

- Verificare differenze locale/remoto
- Vedere i commit che sono **sul remoto** ma **non ancora in locale**
  - git log HEAD..origin/master
  - oppure
  - git log HEAD..origin/main
- Vedere i commit che sono **in locale** ma **non ancora sul remoto**
  - git log origin/master..HEAD
- Se vuoto: NON ci sono commit su origin/master che tu NON abbia già in locale

# GIT REVERT | CREA UN COMMIT CHE RITORNA AD UN COMMIT PRECEDENTE

- git rm nomefile -> Rimuove un file tracciato
- git commit -m "Rimosso cancellare.txt"
- git push

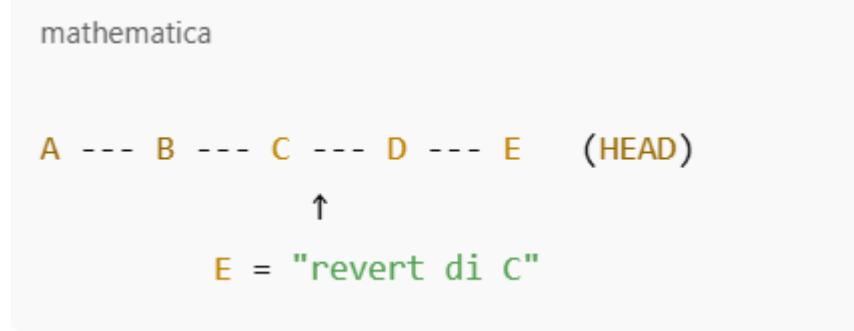
· Il file viene rimosso **dal repository remoto**

· Recuperare un file DOPO commit + push

· git revert (se vuoi annullare la cancellazione)

· Esempio:

· git revert <commit-che-ha-cancellato-il-file>



# GIT PUSH –U ORIGIN MAIN (UPSTREAM) CONDIVIDERE MODIFICHE CON REMOTO

- git push origin branch -> Carica modifiche su un repository remoto

```
git push -u origin main
```

**-u significa upstream: 'upstream è il branch remoto “di riferimento” del tuo branch locale.**

In pratica Git salva questa informazione

```
git push -u origin nomeBranch
```

I push successivi possono essere >git push e git sa già a cosa si deve riferire

## GIT BRANCH -R

- consente di vedere i branch remoti disponibili
- git branch -r

```
● PS C:\Corsi\git\corso-git-2026> git branch -r  
origin/HEAD -> origin/master  
origin/m1  
origin/master
```

# CANCELLARE UN BRANCH REMOTO

- `git push origin --delete nome-branch`
- `git push origin --delete mio-branch`
- Questo elimina il branch dal repository remoto (GitHub).
- Il branch scompare dal remoto
- Gli altri sviluppatori:
  - lo vedranno sparire al prossimo git fetch
  - I branch locali NON vengono cancellati automaticamente

# CONFRONTO DETTAGLIATO LOCALE VS REMOTO

- git log HEAD..origin/main
- git log --oneline --graph --all --decorate

```
git <command> [<revision>... ] [ <file>... ]  
PS C:\Corsi\git\corso-git-2026> git log --oneline --graph --all --decorate  
>>  
* 3c19117 (HEAD -> m1) soluzioni  
* 0095ebe (origin/m1) m1  
* 444bca4 (origin/master, origin/HEAD, master) first
```

## Riga 1

\* 3c19117 (HEAD -> m1) soluzioni

### Significa:

- 3c19117 → hash del commit
- HEAD -> m1 →
  - **sei posizionato su questo commit**
  - il branch locale **m1 punta qui**

• soluzioni → messaggio del commit

👉 **Questo commit esiste solo in locale** (non è ancora sul remoto).

## Riga 2

\* 0095ebe (origin/m1) m1

### Significa:

- origin/m1 → branch remoto
  - il remoto m1 è **fermo al commit precedente**
- 👉 Il branch locale m1 è **avanti di 1 commit** rispetto al remoto.



# CONFRONTO DETTAGLIATO LOCALE VS REMOTO

- git log HEAD..origin/main
- git log --oneline --graph --all --decorate

```
git <command> [<revision>... ] [ <file>... ]  
PS C:\Corsi\git\corso-git-2026> git log --oneline --graph --all --decorate  
>>  
* 3c19117 (HEAD -> m1) soluzioni  
* 0095ebe (origin/m1) m1  
* 444bca4 (origin/master, origin/HEAD, master) first
```

## Riga 3

- \* 444bca4 (origin/master,  
origin/HEAD, master) first

### Significa:

- master → branch locale master
- origin/master → branch master sul  
remoto
- origin/HEAD → branch **predefinito del  
remoto**
- Tutti puntano allo stesso commit (first)  
 master locale e remoto sono  
**perfettamente allineati.**



## GIT FETCH - / GIT PULL | OTTENERE MODIFICHE DAL REMOTO

- git pull origin branch -> Scarica e applica le modifiche dal remoto

```
● PS C:\Corsi\git\repository-test> git pull origin main
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 972 bytes | 121.00 KiB/s, done.
From https://github.com/docentemaurocasa/dei/repository-test
 * branch           main      -> FETCH_HEAD
   bbbf6e7..a7cd2fd  main      -> origin/main
Updating bbbf6e7..a7cd2fd
Fast-forward
```

# GIT REBASE

- **git rebase** serve a spostare i tuoi commit sopra una base più aggiornata, mantenendo una storia lineare
- Ipotizzando di aver creato un branch **feature** quando main era in **B** e che nel frattempo main sia avanzato fino a **D**

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

- Con **git rebase main**:
  - i commit E e F vengono spostati
  - dalla base **B** alla nuova base **D**
  - senza creare merge commit
- 
- Non usare rebase su branch già condivisi

```
git checkout feature  
git rebase main
```

# GIT CHERRY-PICK

- Serve a copiare UNO o più commit specifici su un altro branch

- **Scenario**

- main: A — B — C

- feature: D — E

- **Comando**

- git cherry-pick D

- **Risultato**

- main: A — B — C — D'

- Solo il commit scelto

- Commit copiato (hash diverso)

- Nessun merge, nessun rebase

# PULL REQUEST | RICHIESTA DI AGGIORNAMENTO

- Quando il repository **NON** è tuo, l'unico modo corretto per chiedere un aggiornamento è la **Pull Request (PR)**.
- Una Pull Request è una richiesta al proprietario del repository di integrare le tue modifiche nel suo progetto.
-  **Workflow corretto Team Aziendale (repo NON tuo, ma con permessi)**
- git clone <https://github.com/originale/progetto.git>
- cd progetto
- git switch -c fix-readme
- # fai modifiche
- git add .
- git commit -m "Correzione README"
- git push origin fix-readme
- Poi:
- apri una **Pull Request (direttamente su github)**
- **branch → branch nello stesso repository**

# PULL REQUEST SU FORK O SU REPO CON DIRITTI

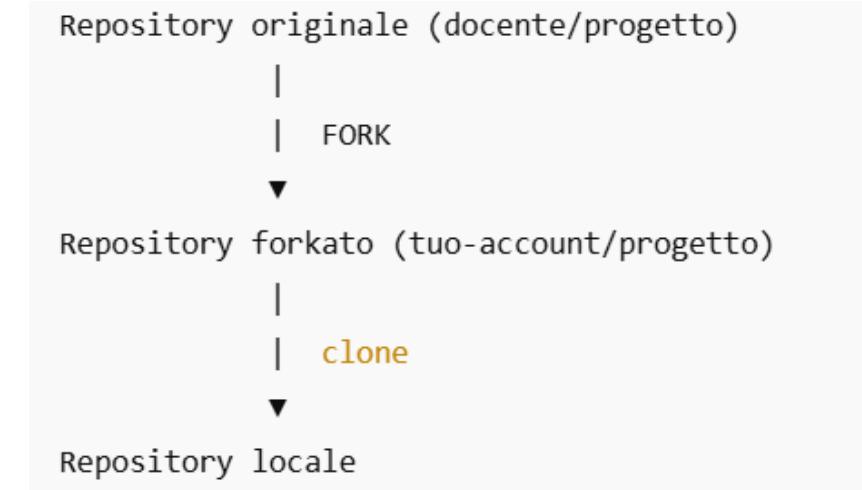
- "PR = lavoro di team"  
"Fork = open source + PR"
- Se ho i diritti su di un repository posso fare un branch nel repository e chiedere una PR, se non ho i diritti posso fare un fork e richiedere una PR al repository originale
- A volte può convenire fare una PR anche se si ha i diritti sul REPO per avere un controllo e una verifica dal team leader
- Una PR risponde a:
  - **perché** è stata fatta questa modifica?
  - **chi** l'ha approvata?
  - **quando** è entrata in main?
- Tutto resta:
  - nei commenti
  - nei commit
  - nella storia del progetto

# FORK | CLONARE UN REPOSITORY NON PROPRIO

- Un fork è un repository remoto indipendente, derivato da un altro repository

-  **Quando usare un fork**

-  Non hai permessi di scrittura
-  Progetti open source
-  Lavoro didattico
-  Collaborazioni esterne



-  **Un fork NON rimane collegato automaticamente al repository originale**
- Quando fai un **fork su GitHub**:
- GitHub crea **un nuovo repository remoto**
- è **indipendente** dall'originale
- **Git non crea alcun collegamento automatico**

# REBASE VS MERGE

## Rebase vs Merge

### Rebase

Storia lineare

Nessun merge commit

Commit riscritti

Per branch personali

### Merge

Storia ramificata

Merge commit

Commit originali

Per branch condivisi

👉 Rebase = pulizia

👉 Merge = sicurezza

# MERGE VS REBASE



```
PS C:\Corsi\git\git-test3> git log --oneline --all --graph
* a98c827 (HEAD -> main) mod main3
* 511804c mod main2
* eaa8a7d mod main1
| * 1d1b7cf (a) mod a3
| * 871659f mod a1
| * fffbd763 mod a1
| * f458552 a
|
* a8194e4 main 1
```

- git checkout main
- git merge a
- \* MBBBBB (HEAD -> main)
- | \
- | \* 1d1b7cf (a) mod a3
- | \* 871659f mod a1
- | \* fffbd763 mod a1
- | \* f458552 a
- \* | a98c827 mod main3
- \* | 511804c mod main2
- \* | eaa8a7d mod main1
- | /
- \* a8194e4 main 1

Commit  
merge



```
PS C:\Corsi\git\git-test3> git log --oneline --all --graph
* a98c827 (HEAD -> main) mod main3
* 511804c mod main2
* eaa8a7d mod main1
| * 1d1b7cf (a) mod a3
| * 871659f mod a1
| * fffbd763 mod a1
| * f458552 a
|
* a8194e4 main 1
```

- git checkout a
- git rebase main
- git checkout main
- git merge a

```
* 1d1b7cf' (HEAD -> main, a) mod a3
* 871659f' mod a1
* fffbd763' mod a1
* f458552' a
* a98c827 mod main3
* 511804c mod main2
* eaa8a7d mod main1
* a8194e4 main 1
```

