Guida a Git

Cos’è Git (in parole semplici)

Git è un **sistema di controllo di versione**:
tiene traccia di tutte le modifiche fatte ai file di un progetto nel tempo.

> Immaginalo come una “macchina del tempo” per il tuo codice:
> puoi tornare indietro a una versione precedente, vedere chi ha fatto un cambiamento, o lavorare in parallelo con altre persone senza sovrascrivere il lavoro altrui.

A cosa serve

- **Salvare la cronologia del progetto**
- **Lavorare in team** in modo sicuro e organizzato
- **Sperimentare** su “rami” separati (branch)
- **Unire** modifiche di più sviluppatori
- **Documentare** lo sviluppo (tramite i messaggi di commit)

1. Installazione e configurazione iniziale

Installazione

- **Windows / macOS:** scarica da git-scm.com
- **Linux:**

  ```
  sudo apt install git
  ```

Configurazione (una sola volta)

Dopo l’installazione, imposta nome ed email per i commit:

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

Puoi controllare la configurazione con:

```
git config --list
```

2. Creare un nuovo repository

Un **repository (repo)** è la cartella “sotto controllo Git”.

Inizializza Git in una cartella

```
git init
```

Questo comando crea una sottocartella nascosta `.git/`
→ contiene tutta la cronologia e le versioni dei file.

3. Aggiungere file e fare il primo commit

Stato del repo

```
git status
```

Mostra i file nuovi, modificati o in attesa di essere salvati.

Aggiungere file da tracciare

```
git add .
```

Aggiunge **tutti** i file (puoi anche fare `git add nomefile.js` per uno solo).

Salvare lo snapshot (commit)

```
git commit -m "Primo commit: inizializzazione progetto"
```

Ogni commit ha un messaggio che descrive il “perché” della modifica.

4. Struttura logica di Git

Git organizza il lavoro in **tre aree principali**:

Area	Nome	Significato
Working Directory	i file su cui stai lavorando	i file reali del progetto
Staging Area	area intermedia	dove prepari i file per il commit
Repository	storico ufficiale	dove vengono salvati i commit

Il flusso tipico:

```
Modifica file → git add → git commit
```

5. Cronologia e versioni

Vedere la cronologia

```
git log
```

Mostra l’elenco dei commit con autore, data e messaggio.

Versione compatta

```
git log --oneline
```

Tornare a una versione precedente

```
git checkout <id_commit>
```

(per tornare indietro temporaneamente)

6. Branch: lavorare in parallelo

Un **branch (ramo)** è una linea di sviluppo indipendente.
Il ramo principale si chiama normalmente `main`.

Creare un nuovo branch

```
git branch feature-login
```

Passare a un branch

```
git checkout feature-login
```

oppure (forma moderna)

```
git switch feature-login
```

Creare e passare in un solo comando

```
git checkout -b feature-login
```

Tornare al branch principale

```
git switch main
```

7. Unire branch (merge)

Quando una funzionalità è pronta, la riportiamo nel ramo principale.

```
git switch main
git merge feature-login
```

> Se i due branch hanno modificato le stesse righe, Git segnalerà un **conflitto**: dovrai decidere quale versione tenere.

8. Risolvere conflitti

Se Git trova differenze incompatibili:

- Apri i file “in conflitto”
- Cerca i segni speciali:

  ```
  <<<<<<< HEAD
  codice del branch principale
  =======
  codice del branch secondario
  >>>>>>> feature-login
  ```

- Tieni la parte giusta, elimina i segni, poi:

  ```
  git add <file>
  git commit
  ```

> Suggerimento: usa Visual Studio Code — riconosce automaticamente i conflitti e propone pulsanti per risolverli.

9. Mettere via modifiche temporaneamente (stash)

Hai modifiche in corso ma devi cambiare branch?
Salvale “da parte”:

```
git stash
```

Quando vuoi riprenderle:

```
git stash pop
```

10. Ignorare file con `.gitignore`

Alcuni file **non devono mai essere tracciati**:

- `node_modules/`
- `.env`
- `uploads/`
- file temporanei o di build

Crea un file chiamato `.gitignore` nella root del progetto:

```
node_modules/
.env
uploads/
dist/
```

> Così Git ignorerà quei file anche se esistono nella cartella.

11. Rimuovere o ripristinare file

Rimuovere un file tracciato

```
git rm nomefile.js
git commit -m "rimuove file non più necessario"
```

Annullare modifiche non ancora aggiunte

```
git restore nomefile.js
```

Annullare modifiche già aggiunte allo stage

```
git restore --staged nomefile.js
```

12. Vedere differenze

```
git diff
```

Mostra **linea per linea** cosa è cambiato rispetto all'ultimo commit.

Puoi confrontare anche due commit:

```
git diff id1 id2
```

13. Taggare versioni

I **tag** servono per segnare momenti importanti (es. versioni ufficiali).

```
git tag -a v1.0.0 -m "Versione iniziale stabile"
git tag
```

14. Collegare un repository remoto (GitHub)

Creare un repo su GitHub

Dai nome e descrizione, **senza README** (lo creerai in locale).

Collegare il repo locale a GitHub

```
git remote add origin https://github.com/tuonome/tuo-progetto.git
```

Inviare i commit al remoto

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

Scaricare modifiche dal remoto

```
git pull
```

> ⚠ Non usare mai `git push --force` a meno che tu sappia esattamente cosa stai facendo — sovrascrive la cronologia remota!

15. Comandi principali riassunti

Operazione	Comando
Inizializza repo	`git init`
Aggiungi file	`git add .`
Salva commit	`git commit -m "messaggio"`
Mostra stato	`git status`
Vedi cronologia	`git log --oneline`
Crea branch	`git branch nome`
Passa branch	`git switch nome`

Operazione	Comando
Unisci branch	`git merge nome`
Ignora file	`.gitignore`
Ripristina file	`git restore nome`
Sincronizza repo remoto	`git push` / `git pull`

16. Buone pratiche

Fai commit **piccoli e frequenti**
Scrivi messaggi chiari:

```
feat: aggiunge rotta di login
fix: corregge validazione email
chore: aggiorna dipendenze
```

Controlla sempre `git status` prima di committare
Non aggiungere mai file sensibili (`.env`, credenziali, token)
Dai nomi chiari ai branch (`feature/auth`, `fix/upload-bug`)

17. Flusso di lavoro tipico (da soli)

1. `git init`
2. `git add .`
3. `git commit -m "inizializza progetto"`
4. `git branch feature/nuova-funzione`
5. `git switch feature/nuova-funzione`
6. Modifica → `git add` → `git commit`
7. `git switch main` → `git merge feature/nuova-funzione`
8. `git push origin main`

18. Flusso di lavoro tipico (in team)

1. Clona repo esistente:

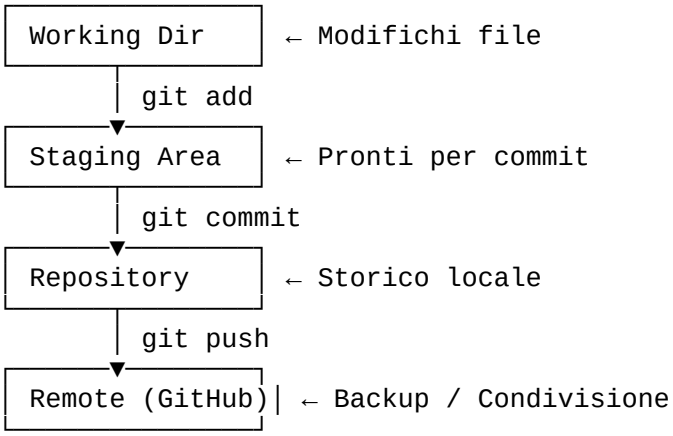
   ```
   git clone https://github.com/org/progetto.git
   ```

2. Crea branch per la tua parte:

   ```
   git checkout -b feature/tuo-task
   ```

3. Lavora → commit → push sul tuo branch
4. Apri una **Pull Request** su GitHub
5. Code review → merge su `main`

19. Git in breve — schema visivo



20. Conclusione

Git non è solo un “salvataggio intelligente”:
è un **modo di pensare lo sviluppo**.
Permette di lavorare in modo **ordinato, reversibile e collaborativo**.

Conoscere Git significa:

- non avere più paura di “rompere tutto”
- poter sperimentare liberamente
- documentare il progresso nel tempo