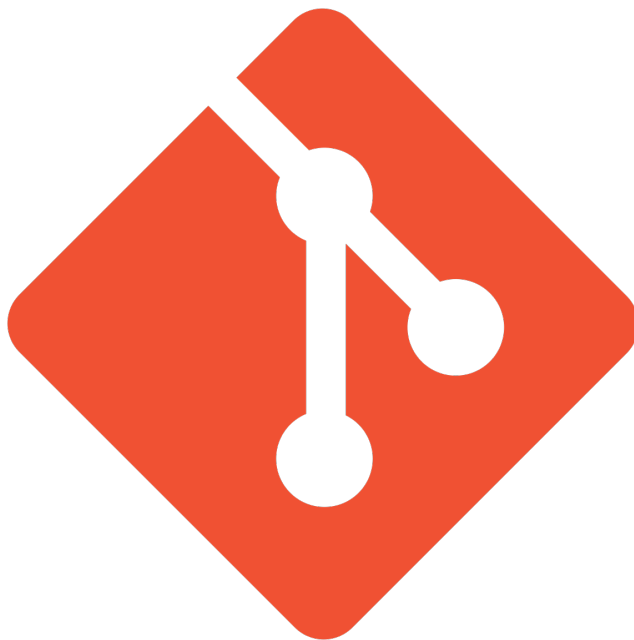


# Git Manual

LuBu





## Sommario

Questo è un manuale introduttivo a GIT scritto durante le lezioni di Metodi e Tecnologie per lo Sviluppo Software, a.a. 2021/22



# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Staging Area . . . . .	3
1.2	Stato di un file in GIT . . . . .	3
1.3	Configurazione . . . . .	3
1.4	Creazione del repository . . . . .	4
<b>2</b>	<b>Comandi</b>	<b>5</b>
2.1	Comandi Base . . . . .	5
2.2	Comandi di stato e ripristino delle modifiche . . . . .	6
2.3	Comandi Branch . . . . .	6
2.4	Repository Remoto . . . . .	6
2.5	Sincronizzazione . . . . .	7
<b>3</b>	<b>Gitignore</b>	<b>8</b>



# 1 Introduzione

Git è un software di controllo versione distribuito utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005.

La maggior parte delle operazioni viene fatta in locale.

Distribuito per “clone” del repository, permette backups multipli e la possibilità di adottare diversi Work flow.

Ogni commit è identificato da un ID (checksum SHA-1 di 40-caratteri basato sul contenuto di file o della struttura della directory) che ne garantisce l'integrità. Non è possibile cambiare un commit senza modificare l'ID del commit stesso e di i commit successivi

## 1.1 Staging Area

È stata aggiunta un'area di staging dove vengono validati i file modificati che potranno essere versionati con un commit.

In GIT i file della copia locale possono essere:

- Nella **Working directory** Checked out, modificati ma non ancora validati (Modified);
- Nella **Staging Area**, validati ma non ancora committati. Il commit salva uno snapshot di tutti i file presenti nella staging area (Staged);
- Nel **Repository locale** (Committed)

## 1.2 Stato di un file in GIT

Un file in GIT può essere in uno dei seguenti 3 stati:

- **Modifed:** modificato nella working directory;
- **Staged:** salva una snapshot nella staging area;
- **Committed:** preso dalla staging area e salvato nel repository locale.

## 1.3 Configurazione

E' richiesta una configurazione iniziale dove vengono impostati l'username e l'email da usare per ogni commit:

- `git config -global user.name "username"`
- `git config -global user.email user_email@example.com`

E' possibile invocare il seguente comando per avere la lista di tutte le configurazioni:

```
git config -list
```

Le configurazioni possono essere fatte a vari livelli:

- **system:** per l'intero sistema per tutti gli utenti
- **global:** per il singolo utente
- **local (di default):** per singolo repository



## 1.4 Creazione del repository

Due scenari:

**Creazione del repository locale nella cartella corrente:**

```
git init
```

Crea una cartella .git nella cartella corrente. Da questo momento è possibile iniziare il versionamento dei file localmente.

**Clonazione di un repository remoto nella cartella corrente:**

```
git clone urlLocalDirectoryName
```

Clona il contenuto del repository remoto nella cartella corrente e crea una cartella .git che rappresenta il repository locale



## 2 Comandi

### 2.1 Comandi Base

- `git init` inizializza la cartella
  - `git add` aggiunge i file nella staging area e crea una snapshot. In particolare:
    - `git add 'nomeFile'` aggiunge il file 'nomeFile' alla staging area (è possibile indicare più file da aggiungere)
    - `git add *.estensione` aggiunge tutti i file con estensione .estensione alla repo git
    - `git add .` aggiunge tutti i file che hanno subito modifiche e i file precedentemente non tracciati alla staging area
  - `git commit -m 'Titolo'` serve ad eseguire il commit e a dare un titolo (posso aggiungere una descrizione con `git commit -m "Titolo" -m "descrizione"`)
  - `git restore -staged 'nomefile'` rimuove il file 'nomefile' dalla repo git
  - `git commit -amend` modifica titolo/descrizione dell'ultimo commit
  - `git reset` serve ad eliminare i commit. In particolare:
    - `git reset --soft` riporta lo stato alla staging area precedente al commit.
    - `git reset -mixed` riporta lo stato precedente alla staging area (default).
    - `git reset -hard` cancella sia il commit che eventuali file aggiunti non presenti ad un commit precedente dalla working directory.
- N.B.** vicino al comando `reset` oltre alla modalità si deve indicare `HEAD` + uno tra `'id-commit'`, `^`, `,`, `~`:
- `'id commit'` riporta lo stato al commit di cui è stato indicato l'id.
  - `^` torna indietro di un commit per ciascun `^` presente.
  - `~#commit` torna indietro di un numero di commit pari a quello indicato dal numero inserito.

#### ESEMPIO:

`git reset -soft HEAD^^` → torna indietro di 2 commit riportando i file non presenti al commit in questione alla staging area.

`git reset -hard HEAD~3` → torna indietro di 3 commit eliminando le modifiche non presenti nel commit al quale si sta tornando.



## 2.2 Comandi di stato e ripristino delle modifiche

- `git status` mostra lo stato dei file all'interno del progetto  
`git status -s` (short version) mostra lo stato dei file all'interno del progetto con una versione più breve
- `git diff` Per vedere cos'è stato modificato ma non ancora validato nella staging area
- `git diff -cached` Per vedere cos'è stato modificato nella staging area
- `git checkout - 'nomeFile'` serve a rimuovere le modifiche dal file 'nomeFile'.
- `git restore 'nomeFile'` serve a rimuovere le modifiche dal file 'nomeFile'. (Stesso comportamento del comando precedente).
- `git log` mostra i conti eseguiti fino a questo momento e le loro informazioni (id, titolo, descrizione, autore, data).  
Si possono utilizzare opzioni quali `-oneline` per delle informazioni più compatte, `-reverse` per vederle in ordine inverso  
`git log -2` Per vedere le ultime 2 modifiche

## 2.3 Comandi Branch

- `git branch` mostra i branch esistenti nel progetto. Il branch su cui ci troviamo al momento corrente è indicato da un \*
- `git branch 'nomeBranch'` serve per creare un nuovo branch con nome 'nomeBranch'
- `git checkout 'nomeBranch'` sposta il puntatore HEAD verso il branch 'nomeBranch', quindi in parole povere serve a spostarsi tra i vari brach.
- `Git checkout -b 'nomeBranch'` crea un nuovo branch 'nomeBranch' e sposta il puntatore HEAD sullo stesso
- `git merge 'nomeBranch'` serve ad unire il branch 'nomeBranch' nel branch corrente
- `git branch -merged` serve a vedere quali branch sono stati uniti
- `git branch -M 'nomeBranch'` serve per rinominare il branch corrente in 'nomeBranch'
- `git branch -d 'nomeBranch'` serve ad eliminare il branch 'nomeBranch'. **Funziona solo se il branch è stato unito ad un altro branch**
- `git branch -D 'nomeBranch'` serve ad eliminare il branch 'nomeBranch'
- `git branch -abort` serve ad interrompere il merge pendente tra due branch

## 2.4 Repository Remoto

- `git remote add 'nomeServer' 'urlServer'` serve per inserire l'indirizzo di un server remoto a cui effettuare push e pull
- `git remote` fornisce la lista dei nomi dei server remoti registrati in precedenza.



- `git remote -v` fornisce la lista dei nomi e degli url dei server remoti registrati in precedenza
- `git remote show 'nomeServer'` Serve ad ispezionare la configurazione del server 'nomeServer'
- `git remote rename 'nomeServer' 'nuovoNomeServer'` serve a rinominare il server 'nomeServer' in 'nuovoNomeServer'
- `git remote remove 'nomeServer'` serve ad eliminare il server

## 2.5 Sincronizzazione

- `git push -u 'nomeServer' 'nomeBranch'` serve per fare l'upload del nostro codice ad un server remoto. L'opzione -u serve a specificare nella configurazione un server remoto di default
- `git fetch` sincronizza le informazioni della repository remota nella repository locale. Dopo è necessario effettuare il comando `git merge` per unire i commit
- `git pull` sincronizza le informazioni della repository remota nella repository locale unendo automaticamente i commit
- `git clone 'urlServer'` clona il remote server 'urlServer' in locale, mantenendo tutti i commit e le modifiche nel tempo

Il comando clone accetta anche alcune opzioni **ad esempio:**

`git clone 'urlServer' 'nomeCartella'` clona il server in locale assegnando il nome 'nomeCartella' alla repository appena clonata





### 3 Gitignore

Il comando `touch .gitignore` crea un file **.gitignore** che ignora i file e le cartelle contenuti in esso dalla funzione di `add` e di conseguenza dai `commit`.

È possibile aprire il file `.gitignore` con un **editor di testo**.

All'interno del file `.gitignore` è possibile scrivere:

- **commenti a linea singola** iniziando la riga con il carattere `#`
- **nomi dei file/cartelle** che si desidera ignorare
- **\*.estensione** per rimuovere tutti i file con l'estensione indicata  
!'`nomeFile`' permette di rendere visibile un file la cui estensione è indicata all'interno del file `.gitignore` di cui però si vuole mantenere la visibilità

#### Esempio di file `.gitignore`

```
#directories
ignorami
ignorami_due

#Files
ignorami.jpg
ignorami.png
ignorami.txt
ignorami.mp3

#Extensions
*.gz
*.aux
*.fdb_latexmk
*.fls
*.out
*.toc
*.log

#macOS
.DS_Store
```