

GIT Per Laboratorio

Jacopo Basso Ricci

March 15, 2026

Indice

1	Introduzione	2
1.1	Minima storia	2
2	Architettura	2
2.1	Locale	2
3	Setup	3
4	Status	4
5	Staging	4
6	Committing	5
7	Rimuovere file	5
8	Invertire i cambiamenti	5
9	Log	5
10	Branching	6
10.1	Creazione di Branch	6
10.2	Muoversi tra i Branch	6
10.3	Merge	7
10.3.1	Merge Conflit	7
10.4	Muoversi nel Tempo	7
11	Mostrare le differenze	8
12	Finalmente online	8
13	Push	8
14	Fetch	8
15	Quality of life	9
15.1	Stash	9
15.2	.gitignore	9



1 Introduzione

git: the stupid content tracker

Questa è la definizione data nella documentazione ufficiale di cosa sia git e tanto più lo esporrete tanto più capirete perché si definisce in questo modo. Tutto quello che fa questo programma è gestire le versioni dei file tramite un'interfaccia che vuole provare a essere semplice. Sotto la scocca altro non è che un database.

Nota importante: git non è gitHub. Git lavora in locale, sui vostri computer, mentre gitHub è un sito web che si offre come fornitore di server git. Diciamo che git è un file che voi avete sul vostro computer, gitHub è il Google delle repository git.

Qui ci concentreremo su un insieme ristretto di comandi e concetti che sono utili.

1.1 Minima storia

Partiamo subito con qualcosa di inutile.

Git nasce per la gestione delle versioni del Kernel Linux nel 2005. Prima di quella data lo sviluppo del Kernel era fatto usando BitKeeper, ma un maintainers legato al progetto di Rsync fece reverse engineering sul protocollo di comunicazione proprietario dell'applicazione. Di tutta risposta quelli di BitKeeper buttarono fuori gli sviluppatori di Linux dal loro sistema lasciandoli senza un modo per fare tracking delle versioni. Così in 5 giorni Linus Torvald (il gentile dittatore di Linux) scrisse git.

Molta della filosofia di git nasce per gestire il Kernel che è un progetto mastodontico paragonato a quello che normalmente fa un gruppo di Laboratorio, ma proprio perché è pensato per grandi progetti è estremamente efficace nel gestire moltissimi casi particolari.

2 Architettura

L'unità fondamentale di git è la repository che è il luogo che contiene tutte le versioni dei tuoi file. Si può pensare che sia una libreria dove tutto quello che viene versionato è catalogato e salvato.

Nel suo funzionamento di gestore di repository git si può dividere in 2 parti fondamentali: la componente locale e quella remota.

Come detto la parte locale vive sul proprio sistema mentre quella remota su un server che useremo per comunicare.

2.1 Locale

Localmente il sistema è ulteriormente suddiviso in 3 componenti:

- Working Directory
- Stage
- Local Repository

La Working Directory non è altro che la cartella dove lavorate e tutte le sottocartelle.

Lo Stage è più difficile da capire intuitivamente, ma è un'area temporanea dove i file aspettano tra lo stadio di attivo lavoro e un salvataggio dentro una versione locale.

La Local Repository è il luogo dove si salvano tutte le versioni che si vogliono mantenere sul proprio computer. Idealmente è una fotografia del vostro codice che viene salvata dentro il vostro personale sistema di versionamento.



Spesso si usa l'analogia di una persona che si veste per andare ad una festa e si dice che la Working Directory è il bagno dove una persona si pulisce e poi si veste. La fase di Stage è lo specchio dove si cerca di capire se si è abbastanza eleganti e pronti a uscire. Infine la Local Repository è una foto perfetta di come siete che vi permette di ricordarvi esattamente come siete vestiti. Andare alla festa è dunque il passaggio dalla zona locale a quella remota.

3 Setup

Ignorerò la parte di installazione dicendovi di andare su un qualunque browser e cercare come si installa per il vostro sistema operativo. Ricordo che avere git sul sottosistema Linux per Windows non credo installi automaticamente git sull'intero sistema operativo così come viceversa.

Per testare che git sia installato correttamente lanciare:

```
1 git --version
```

Se l'installazione è andata a buon fine non ci saranno errori e leggerete la versione di git che state utilizzando.

Ora create una nuova cartella ed entratevi, lì lanciate il comando:

```
1 git init
```

Vedrete che il comando vi risponde che è stata inizializzata una nuova repository vuota. All'atto pratico l'unica cosa che vederete è la creazione di una cartella nascosta che si chiama *.git*. Tutto quello che git saprà della vostra repository vivrà in questa cartella, quindi cancellarla eliminerà ogni versione salvata della vostra repository locale.

Per completare le operazioni di setup è necessario lanciare 2 ulteriori comandi che servono a specificare chi effettua le modifiche.

```
1 git config --global user.name "YourName"  
2 git config --global user.mail "you@example.com"
```

Questo imposta la tua identità che è necessaria sia per fare commit (vedremo successivamente) che per caricare codice sul server. Se si vuole impostare l'identità solo per questa repository e non per tutto l'utente baste eliminare *global* e aggiungere *local*.

Questo comando non è strettamente obbligatorio in generale, ma è preferibile:

```
1 git config --global init.defaultBranch main
```

Stiamo dicendo a git che vogliamo impostare globalmente il nome del branch (concetto successivo) principale a main.

Siamo quindi pronti per utilizzare git.



4 Status

Create alcuni file dentro la vostra Working Directory (non la cartella `.git` quella non verrà mai toccata) e scriveteci dentro del testo. Lanciare:

```
1 git status
```

Questo comando è il primo che vedremo che dobbiamo conoscere perché ci permetterà di sapere cosa sta facendo git e qual è lo stato dei file nella nostra repository. Ci sono alcuni stati che sono particolarmente importanti:

untracked: git non ha dati su quel file

staged: pronto per il prossimo commit

committed: salvato dentro il versionamento della repository

Come si può notare gli stati sono legati alla struttura locale: un file `untracked` vive solo nella vostra Working Directory, un file `staged` nello stage e un file `committed` è parte della repository. Notare che un file `untracked` se rimosso sparirà per sempre perché git non ha l'ha mai neppure letto.

5 Staging

Ora noi vogliamo spingere i file dentro la fase di staging. Per farlo useremo il comando:

```
1 git add
```

Il comando di per se restituisce un errore perché non sa cosa aggiungere, e se proverete a lanciarlo vi darà un errore chiedendo se si voleva scrivere:

```
1 git add .
```

Proprio come in molti altri comandi Linux aggiungere il `.` specifica che si vuole lanciare il comando su tutta la cartella. In questo caso verranno automaticamente aggiunti alla fase di staging tutti i file a partire dalla cartella corrente in modo ricorsivo, quindi anche tutti i file nelle sottocartelle.

Questo comando può essere usato in modo più capillare e preciso con la sintassi:

```
1 git add nomeFile.ext
```

Questo porterà alla fase di staging soltanto il file selezionato (si può usare TAB per autocompletare il nome).

L'ultima variazione del comando presentata è la più potente:

```
1 git add -A
2 git add --all
```

Queste 2 righe, che sono del tutto equivalenti (git le legge come lo stesso comando, uno è l'abbreviazione dell'altro), aggiungono tutti i file che sono stati modificati alla fase di staging. A differenza della versione con `.` questo comando non si limita alla cartella corrente e alle sottocartelle, ma funziona su tutte le cartelle dentro la repository.

Se lanciate il comando `git status` vedrete che i file sono stati spostati alla fase di staging e vi verrà detto quale comando lanciare per togliere i file dalla lista di quelli in staging (dopo).



6 Committing

Dopo la fase di staging vogliamo spingere i file dentro la nostra repository locale, il git questo si fa con il comando:

```
1 git commit -m "Testo del commit, molto utile"
```

Il `git commit` sarebbe sufficiente, ma è consigliabile aggiungere un commento per chiarificare cosa sia stato fatto e rendere più maneggevole il movimento dentro l'albero dei commit. La flag `-m` (messages) è seguita da un messaggio contenuto in doppi apici.

Il comando restituisce una piccola spiegazione delle modifiche che sono state inserite e un hash corto che sarà il nome proprio di quel commit. Ora `git status` dirà che non ci sono modifiche dentro la Working Directory e che tutto è stato salvato correttamente.

7 Rimuovere file

Date le nostre conoscenze per rimuovere un file e fare staging è necessario:

```
1 rm fileName.ext
2 git add fileName.ext
```

Questo viene semplificato dalla funzione

```
1 git rm fileName.ext
```

Il comando `git rm` è concettualmente l'opposto di `git add`. Proprio come il comando `rm` deve essere usato con cautela.

L'altra grande funzionalità di `git rm` si ha quando viene combinato con la flag `-cached`:

```
1 git rm --cached fileName.ext
```

Questo comando toglie i file da quelli in fase di staging.

8 Invertire i cambiamenti

Ipotizziamo che tu abbia fatto lo staging di uno o più file e dopo li abbia modificati nella Working Directory. Ora si vuole riportare la versione nell'ultimo commit dentro la propria Working Directory eliminando la versione su cui si stava lavorando (questo può succedere se un breve test è andato male). Questo può essere fatto con:

```
1 git restore fileName.ext
```

Questo comando elimina automaticamente il file dalla zona di staging.

9 Log

Ora abbiamo tutti gli strumenti per poter creare commit, ma come capiamo cosa abbiamo fatto? Esattamente come abbiamo `git status` per capire a che punto siamo della fase di staging abbiamo:

```
1 git log
```

per comprendere cosa abbiamo fatto dei nostri commit. `git log` mostra per ogni commit l'hash, che è il nome interno a git, autore, orario e il commento di commit. Si può usare il tag *oneline* per vedere un commit per riga. Il comando diventa:

```
1 git log --oneline
```

In questa forma l'hash mostrato è la versione ridotta che può essere utilizzata come nome completo di un commit anche nei comandi che vedremo poi.



10 Branching

Questo è il concetto più difficile di tutto il funzionamento di git. Noi per ora abbiamo un albero dei commit che è lineare: ogni commit arriva dopo il precedente, ma non è sempre il caso. Immaginiamo di voler testare delle modifiche prima di caricarle direttamente. Git ci aiuta creando un ramo diverso di sviluppo. Il codice in questo ramo segue una linea parallela a quella del branch principale: main. Quando vogliamo che un ramo torni nello sviluppo principale faremo un merge: un unione.

In questo paragrafo i concetti da portare a casa sono 3: possiamo creare una ramificazione del nostro sviluppo, queste ramificazioni si comportano in modo completamente analogo al sistema che abbiamo visto prima, e possiamo unire 2 rami di sviluppo separati con l'operazione di merge.

Con il comando

```
1 git branch
```

possiamo mostrare tutti i branch esistenti. Fino ad ora non abbiamo ancora creato nessun branch quindi vedremo solo il main. Notare che il main ha un asterisco perché attualmente stiamo lavorando soltanto in main.

10.1 Creazione di Branch

In modo molto intuitivo basta scrivere:

```
1 git branch branchName
```

Con lo stesso comando seguito da un nome (che non deve essere di un altro branch) noi creiamo un nuovo ramo. Questo comando non modifica la propria Working Directory, né il branch in cui siamo. Lanciando `git branch` vedremo ora che ci sono 2 branch e che noi siamo ancora nel main.

10.2 Muoversi tra i Branch

Per spostarsi tra i branch useremo:

```
1 git checkout branchName
```

Questo ci permette di spostarci tra i rami della repository soltanto se non ci sono modifiche che sono pending o in fase di staging. Dopo introdurremo la stash che serve a evitare di dover fare un commit per poter tornare ad uno stato in cui il comando `git status` ci dice che non c'è niente da fare, per ora basta fare un commit.

Ricapitolando

Per ora sappiamo come:

visualizzare i branch: `git branch`

creare branch: `git branch nomeBranch`

muoverci tra i branch: `git checkout nomeBranch`

Immaginiamo di esserci mossi dal branch main per andare a un nuovo branch chiamato qwerty. Lì abbiamo fatto diverse modifiche, le abbiamo passate alla fase di staging e infine abbiamo creato diversi nuovi commit. Come possiamo importare le modifiche dentro il nostro main?



10.3 Merge

Ora torniamo al nostro branch `main` `git checkout main` o quello che vogliamo contenga l'altro. Adesso possiamo legare i nostri rami.

```
1 git merge nomeBranch
```

Questo potente comando porta tutte le modifiche fatte dentro a `nomeBranch` al ramo in cui siamo, ma non elimina il branch che abbiamo usato per le modifiche.

10.3.1 Merge Conflict

Non sempre tutto è perfettamente funzionante quando tentiamo un merge. Se modifichiamo la stessa sezione in 2 branch diversi e poi tentiamo di unirli git non sa che fare e questo genera un conflitto che deve essere ripulito a mano.

Cerchiamo di capire nel dettaglio come creare un conflitto. Creiamo un branch 1, li scriviamo il file X e scriviamoci qualcosa dentro. Ora facciamo commit di queste modifiche. Facciamo la stessa cosa sul branch 2: creiamolo, generiamo il file X nella stessa posizione e poi lanciamo un commit.

Ora siamo pronti per vedere un merge conflict lanciando in uno dei due branch `git merge branch1` (immaginando di essere nel 2). Il comando restituisce un errore chiedendo intervento umano.

Risolvere un merge conflict è concettualmente molto semplice: basta aprire un editor di testo e leggere quale parte del file ha subito un conflitto. L'errore si presenta in questo modo:

```
1 <<<<<< HEAD
2 codice scritto nel branch corrente
3 =====
4 codice scirtto nel branch di cui si vuole fare merge
5 >>>>>> nomeBranch sotto merging
```

Per pulire il conflitto basta eliminare tutti i decoratori (i simboli aggiunti da git) e scrivere quello che si vuole mantenere. Potremmo voler tenere la versione che già avevamo e quindi eliminare tutto quello che riguarda il branch che stà venendo legato, oppure volere la versione in merging. Non è però necessario tenere una delle versioni presenti ed è possibile scrivere una soluzione custom.

Risolto il conflitto, come consiglia il comando di merge andato male, si deve fare commit della soluzione. Sarà quindi necessario fare ad esempio:

```
1 git add -A
2 git commit -m "Merge Conflit risolto"
```

10.4 Muoversi nel Tempo

Ora che abbiamo le fondamenta di quello che possiamo fare con il branching usiamolo per spostarci tra commit. Sappiamo che con `git log --oneline` possiamo ottenere gli hash ridotti dei nostri commit, che sono i nomi dati da git a quelle versioni. Ora con:

```
1 git checkout hash
```

possiamo muoverci a quel specifico commit come se avessimo riportato il tempo indietro nella nostra Working Directory.

In questo stato noi possiamo creare nuovi commit, ma saremo in uno stato di *detached HEAD*. Essenzialmente quelle modifiche non fanno parte di nessun ramo, in gergo si dice che il commit è orfano. Per risolvere il problema basta modernamente si usa:

```
1 git switch -c nuovoNomeBranch
```

In un colpo solo viene creata una testa per il nostro commit e ci si sposta in essa.



11 Mostrare le differenze

Per capire cosa è stato effettivamente modificato tra un commit ed un altro, o tra 2 branch è possibile usare:

```
1 git diff base modificato
```

base e modificato non sono altro che placeholder. La base è ciò che viene considerata il punto di partenza, mentre il modificato è quello che "importa" le modifiche.

12 Finalmente online

Ora che abbiamo capito come funziona git in locale è il momento di spingerci al di fuori della nostra macchina. Con tutto quello che già sappiamo sarà molto facile entrare nell'ottica di comunicazione con il server online.

Qui non discuteremo di come connettere la repository online a quella locale perché dipende leggermente da fornitore a fornitore.

Ci sono 3 concetti fondamentali:

- push
- fetch
- Pull Request (inutili nel nostro caso)

13 Push

Push è il comando che ci permette di spingere codice sul server. Per poterlo fare è però necessario che la repository locale sia aggiornata e che abbia avuto delle modifiche. Inoltre si deve essere nel branch di cui si vuole fare push. Per essere nel branch di cui si vuole inviare copia al server basta `git checkout nomeBranch`.

Il comando completo:

```
1 git push origin nomeBranch
```

14 Fetch

Fetch serve a richiamare le modifiche dal server. Dopo un fetch la versione più aggiornata del server vive sulla propria macchina in locale, ma non è ancora stato inserito dentro la nostra Working Directory.

```
1 git fetch
```

Fetch serve solo a fare download dell'albero sul server, ma non lo installa. In generale fetch lanciato in questo modo: senza specificare un origine specifica scarica tutti i branch remoti. Per vederli basta:

```
1 git branch -r
```

Come prima il comando serve a vedere i branch e la flag `r` mostra quelli remoti. Per portare la versione più recente sul nostro sistema bisogna fare:

```
1 git merge
```




In questo caso non è necessario specificare quale branch va unito, ma basta lanciare il questo modo il comando e git saprà automaticamente portare le modifiche remote nel proprio sistema. Esiste anche un comando che unisce i 2 passaggi in uno unico:

```
1 git pull
```

Sotto la coscocca quello che fa è chiamare in sequenza **fetch** e **merge**.

Va fatto notare che come prima abbiamo visto i merge conflict, anche in questo caso possiamo ottenere la stessa situazione se 2 persone lavorano contemporaneamente alla stessa sezione dello stesso file, una lo carica sul server e l'altra poi tenta di effettuare un **pull**. I conflitti si risolvono esattamente come discusso prima.

15 Quality of life

In questa sezione spiegherò delle piccole cose che rendono l'utilizzare git estremamente comodo.

15.1 Stash

Immaginiamo che tu stia lavorando ad un tuo commit, e qualcuno ti chieda di andare a visionare un altro branch. Prima proponevo la soluzione di fare un commit, ma quella non ha tanto senso per quanto sia efficace. Tendenzialmente ogni commit dovrebbe essere giustificato e quella di andare a visionare un altro branch non è una giustificazione molto forte. Per superare i limiti di **git checkout** noi vogliamo che le nostre modifiche non ancora dentro un commit vengano messe da parte fino al nostro ritorno.

```
1 git stash
```

Mette tutte le modifiche correnti dentro lo stash, che è una zona di memoria separata da quelle che abbiamo visto prima, riportando la nostra Working Directory allo stato del precedente commit. In questo modo è possibile muoversi liberamente tra i vari branch. Quando però torniamo al nostro ramo noteremo che i file sono ancora alla versione dell'ultimo commit. Per riportare il nostro lavoro indietro dallo stash basta fare:

```
1 git stash pop
```

Note: lo stash è molto più potente di questo singolo caso ed è possibile usarlo come "cestino sicuro" delle proprie modifiche. Infatti è possibile mettere nello stash diverse modifiche e non solo una, ma la trattazione di git che stò facendo mi sembra già abbastanza esaustiva per quello che dobbiamo fare.

Sfortunatamente non credo che esista un comando build-in per effettuare stash automatico al cambio e poi riportarlo nella Working Directory.

15.2 .gitignore

Immaginiamo che stiate facendo un progetto con \LaTeX , la compilazione genera molti file inutili, come: ***.pdf**, ***.log**, ***.out**, ... Poiché non vogliamo caricare tutte queste cose che a gli altri daranno solo fastidio, con le conoscenze a nostra disposizione, dovremmo stare molto attenti a usare:

```
1 git add nomeFile.ext
```

Questa cosa non è molto pratica e ci farebbe perdere un sacco di tempo soprattutto se lavoriamo con file diversi.

Per ovviare a questo problema basta creare un file chiamato **.gitignore**: In questo file si può specificare riga per riga i file da ignorare. Esistono generatori online per crearli in modo procedurale a seconda di cosa bisogna fare.