Git

▼ Che cos'è Git?

Git è il sistema di controllo delle versioni più utilizzato al mondo. Git è un progetto open source originariamente sviluppato nel 2005 da Linus Torvalds, il famoso creatore del kernel del sistema operativo Linux. Git è un esempio di DVCS (Distributed Version Control System), quindi, la copia di lavoro del codice di ogni sviluppatore è anche un repository che può contenere la cronologia completa di tutte le modifiche.

In altre parole, è un sistema che tiene traccia dei cambiamenti ai file del nostro progetto col tempo. Ci permette di registrare cambiamenti al progetto e tornare indietro ad una versione specifica del file tracciati, in qualsiasi momento. Questo sistema può essere usato da molte persone per lavorare insieme in modo efficiente e collaborare su progetti di gruppo, in cui ogni sviluppatore può avere la propria versione del progetto, distribuito sul loro computer. Più tardi, queste singole versioni del progetto potranno essere unite e adattate all’interno della versione principale.

Praticamente, è uno strumento molto conosciuto per coordinare il lavoro parallelo e gestire progetti fra singoli individui e gruppi. Sapere come usare Git è una delle caratteristiche più importanti per ogni sviluppatore oggi giorno.

▼ Benefici principali di Git

- **Ti consente di tenere traccia delle tue modifiche**

  Hai sempre uno stato che monitora esattamente quali modifiche sono state apportate in qualsiasi momento.

- **Backup storico, che usa gli “snapshots” (istantanee)**

  Fondamentalmente puoi mantenere una versione precedente e ripristinarla se necessario. Quindi se vengono riscontrati bug o si incasina qualcosa, è sempre possibile tornare indietro con un backup storico e annullare tali modifiche.

- **Team development**

  Consente a un team di sviluppatori di lavorare effettivamente sullo stesso codice contemporaneamente e quindi unire le loro modifiche insieme per fare progressi su un progetto.

- **È molto flessibile**

  Consente di lavorare localmente su un progetto oppure Git o GitHub possono essere utilizzati come parte dei flussi DevOps.

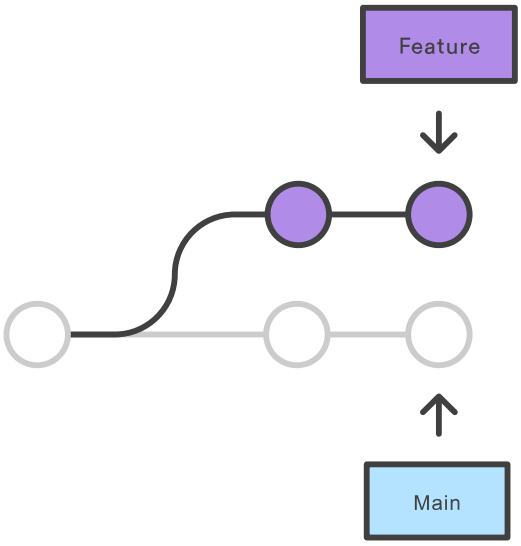
- **Branching development**

  I rami vengono creati dal progetto di base, al fine di sviluppare una nuova funzionalità, correggere un bug o semplicemente eseguire un refactoring. Impediscono agli sviluppatori di software di disturbarsi a vicenda e consentono loro di lavorare in parallelo. Una volta terminata e verificata una modifica, il ramo viene nuovamente unito al tronco.

▼ Git per sviluppatori

Flusso di lavoro del ramo di funzionalità

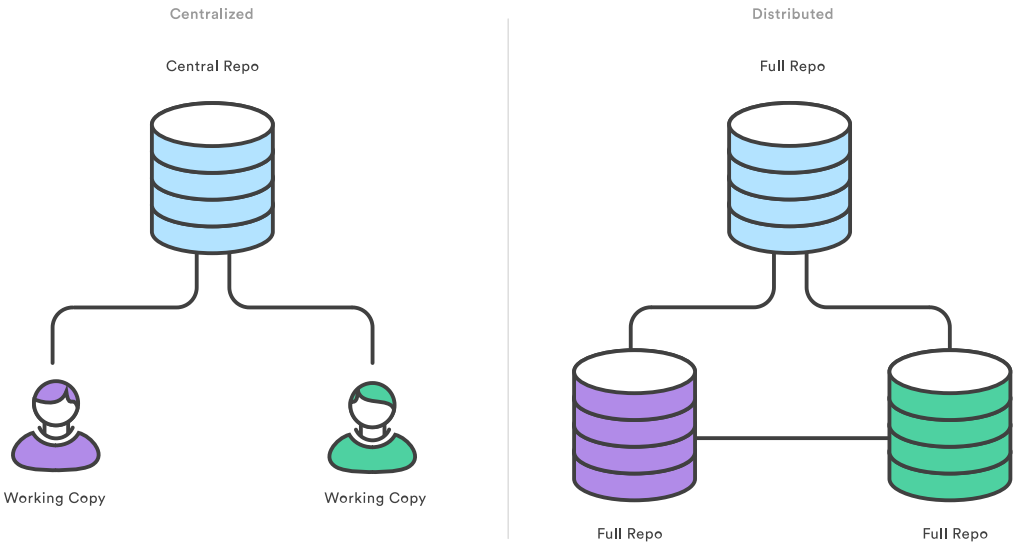
Uno dei maggiori vantaggi di Git sono le sue capacità di ramificazione. A differenza dei sistemi di controllo delle versioni centralizzati, i rami Git sono economici e facili da unire. Ciò facilita il flusso di lavoro degli utenti di Git.



I rami forniscono un ambiente isolato per ogni modifica alla base del codice. Quando uno sviluppatore vuole iniziare a lavorare su qualcosa, non importa quanto grande o piccolo, crea un nuovo branch. Ciò garantisce che il branch principale *(Main)* contenga sempre un codice originale.

Sviluppo distribuito

Git, essendo un sistema di controllo di versioni distribuito, invece di fare una copia della repository, ogni sviluppatore ottiene il proprio repository locale, completa con tutta la cronologia dei commit.

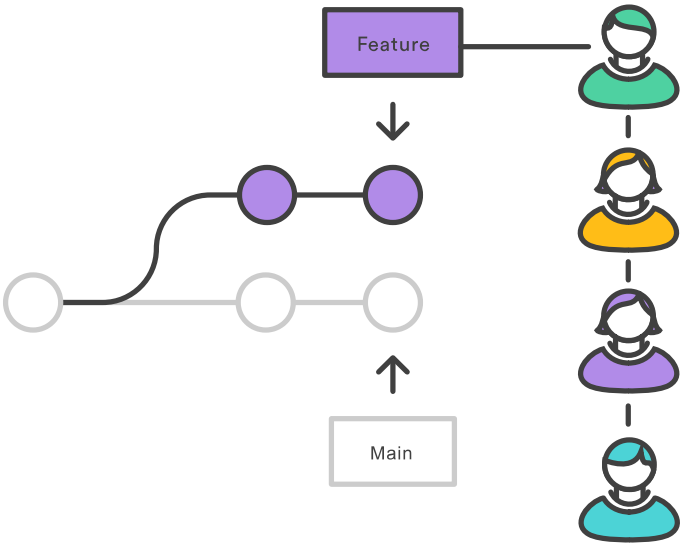


Avere una cronologia locale completa, rende Git veloce, poiché significa che non è necessaria una connessione di rete per creare commit, ispezionare versioni precedenti di un file o eseguire “*diffs*” tra i commit.

Lo sviluppo distribuito semplifica inoltre la scalabilità del team di sviluppo. Se qualcuno interrompe il ramo di produzione in SVN, gli altri sviluppatori non possono archiviare le modifiche finché non vengono risolte. Con Git questo tipo di blocco non esiste. Tutti possono continuare a svolgere le proprie attività nei propri repository locali.

Richieste di pull

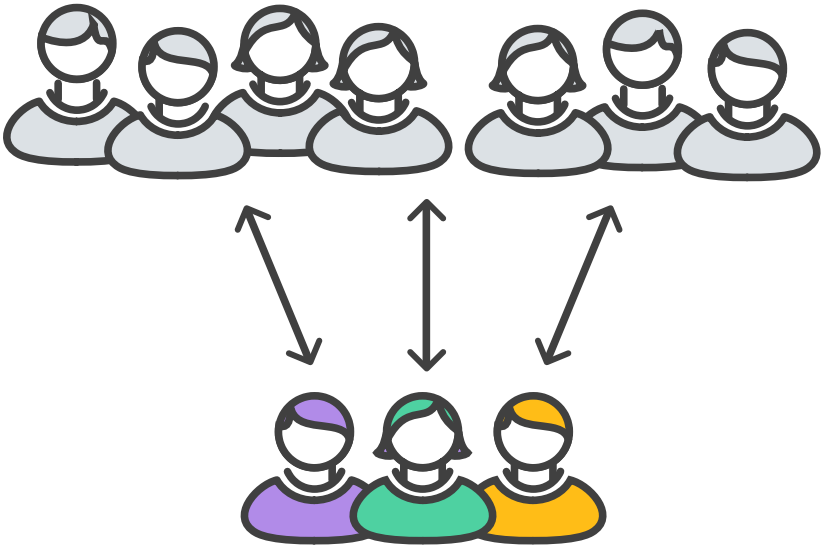
Una richiesta pull è un modo per chiedere a un altro sviluppatore di unire uno dei tuoi rami nel proprio repository. Ciò non solo rende più facile per i responsabili del progetto tenere traccia delle modifiche, ma consente anche agli sviluppatori di avviare discussioni sul loro lavoro prima di integrarlo con il resto della base di codice.



Le richieste pull sono estremamente versatili, infatti, quando uno sviluppatore rimane bloccato con un problema difficile, può aprire una *pull request* per chiedere aiuto al resto del team. Facendo ciò, gli sviluppatori junior possono essere sicuri di non distruggere l'intero progetto trattando le richieste pull come una revisione formale del codice.

Comunità

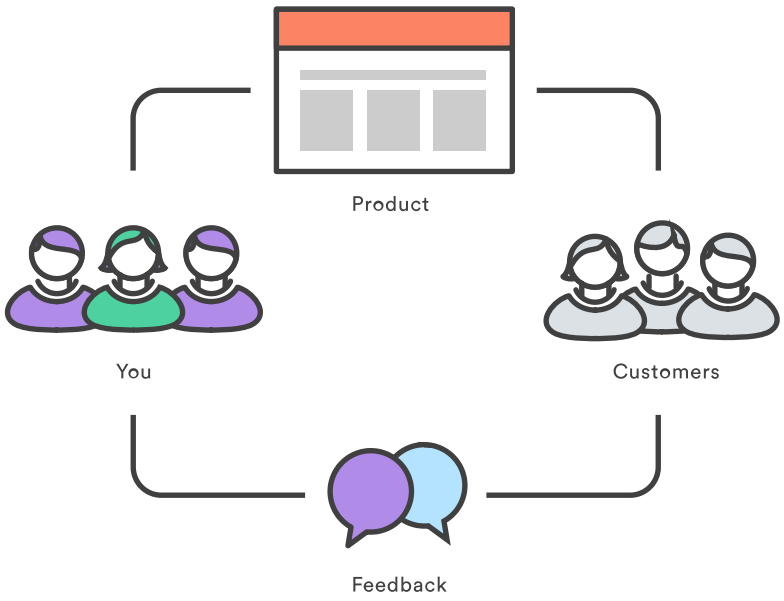
In molti ambienti, Git è diventato il sistema di controllo standard della versione previsto per i nuovi progetti.



Inoltre, Git è molto popolare tra i progetti open source. Ciò significa che è facile sfruttare librerie di terze parti e incoraggiare altri a creare il proprio codice open source.

Ciclo di rilascio più veloce

Il risultato finale di rami di funzionalità, sviluppo distribuito, richieste pull e una comunità stabile è un ciclo di rilascio più rapido. Queste funzionalità facilitano un flusso di lavoro agile in cui gli sviluppatori sono incoraggiati a condividere più frequentemente modifiche più piccole. A loro volta, le modifiche possono essere inviate alla pipeline di distribuzione più velocemente rispetto alle versioni monolitiche comuni con i sistemi di controllo delle versioni centralizzati.



▼ Configurare Git

Come ho detto prima, Git è un VCS. Per usare git, dobbiamo installarlo sul nostro computer.

Installazione

Visita qui per installare Git sul tuo PC (Windows/Mac/Linux). Per verificare se Git è stato installato correttamente, apri il tuo terminale ed esegui questo comando.

```
git --version
```

Dovresti vedere qualcosa del genere.

```
lucaprotelli@Air-di-Luca ~ % git --version
git version 2.32.0
```

Configurazione

Prima di configurare Git sul tuo PC, visita github.com e crea un account.

Ora apri il tuo terminale ed esegui questi comandi.

```
git config --global user.name "nome del profilo GitHub"
git config --global user.email "email del profilo GitHub"
git config --global color.ui
```

▼ Struttura di Git

L’Head

L’Head è il modo in cui git sa in quale ramo ti trovi attualmente. È una specie di collegamento al ramo*(branch)* su cui stai attualmente lavorando.
È un puntatore, quindi punta al nome del branch corrente, ma può puntare anche a un commit specifico (detached HEAD).
Il tuo HEAD si sposterà quando esegui un commit nel ramo attualmente attivo o quando effettui il checkout in un nuovo ramo.

Git tiene traccia dei cambiamenti attraverso tre fasi.

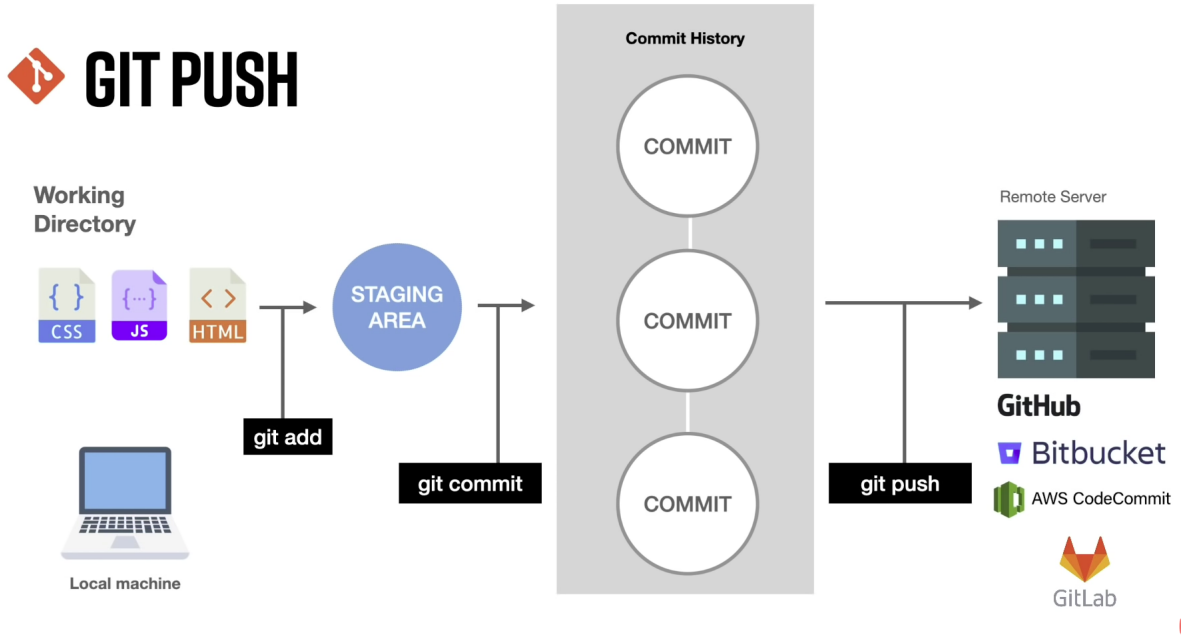
1. Working Directory
2. Index/Staging Area.
3. Commit History*(repo)*.

La working directory (copy) è la cartella del tuo progetto in cui si trovano tutti i tuoi file. Questo non è tracciato da git. È necessario aggiungerli alla Staging Area utilizzando il comando `git add`. Questo processo di aggiunta delle modifiche alla Staging Area dice a git che vogliamo tenere traccia di tutte le modifiche che abbiamo fatto nella nostra directory di lavoro.

Per aggiungere queste modifiche come versione ufficiale, dobbiamo confermare queste modifiche all'Head utilizzando il comando `git commit`. Facendo ciò, le nostre modifiche sono ora salvate sul nostro computer.

Per aggiungere queste modifiche al nostro repository remoto, dobbiamo inviarle a Github utilizzando il comando `git push`. Se imposti il tuo progetto utilizzando `git clone`, saprà già dove inviarle, ma se configuri il tuo progetto utilizzando il comando `git init`, devi creare un repository vuoto su Github e aggiungerlo come repository remoto utilizzando il comando `git remote add origin PASTE_SSH_URL`.

Ora puoi inviare le modifiche utilizzando il comando `git push`.



Come usare Git?

Per usare Git, devi inizializzare un repository git vuoto nella cartella del tuo progetto o clonare un repository esistente da GitHub.

1. Cos’è una repository?

Una repository è un archivio virtuale del tuo progetto. Ti consente di salvare versioni del tuo codice, a cui puoi accedere quando necessario.

Possiamo individuare due maggiori tipi di repositories:

- **repository locale** - un repository isolato salvato sul tuo computer, dove puoi lavorare sulla versione locale del tuo progetto.
- **repository remota** - generalmente salvato al di fuori del tuo sistema locale isolato, di solito su un server remoto. Spesso è utile per lavorare in gruppo, è il posto in cui puoi condividere il tuo codice, vedere quello di altre persone e integrarlo con il tuo e inoltre puoi effettuare un “push” delle modifiche alla repository remota.

---

2. Inizializzazione di una repository

2.1 git init

Usa il comando `git init` per convertire un progetto esistente senza versione in un repository Git o inizializzare un nuovo repository vuoto.

```
lucaprotelli@Air-di-Luca ~ % /Desktop/test$ git init
Initialized empty Git repository in /home/lucaprotelli/Desktop/prova/.git/
```

La maggior parte degli altri comandi Git non sono disponibili al di fuori di un repository inizializzato, quindi questo è solitamente il primo comando che esegui in un nuovo progetto.

2.2 git clone

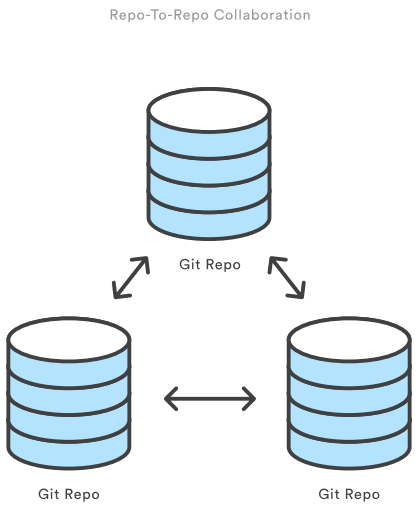
Git clone viene utilizzato principalmente per puntare a un repository esistente e creare un clone o una copia di quel repository in una nuova directory, in un'altra posizione.

Per farlo, vai su GitHub e apri una repository, quindi copia l'URL SSH. Dopodiché esegui questo comando

```
git clone SSH_URL
```

Questo aggiunge la cartella .git e il repository remoto.

Il modello di collaborazione di Git si basa sull'interazione tra repository e repository. Invece di controllare una copia funzionante nel repository centrale, puoi eseguire il *push* o il *pull* dei commit da una repository all'altra.



2.3 git init vs git clone

`git init` e `git clone` possono essere facilmente confusi. Ad alto livello, possono essere usati entrambi per "inizializzare un nuovo repository". Tuttavia, *git clone* dipende da *git init*. *Git clone* viene utilizzato per creare una copia di un repository già esistente in remoto.

---

3. Salvare le modifiche in Git

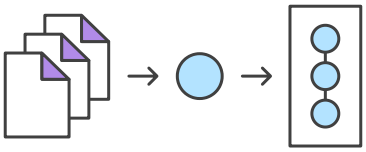
Eseguire un Commit è il processo in cui i cambiamenti sono ufficialmente aggiunti alla repository. In Git, possiamo considerare un commit come un salvataggio o snapshot del tuo progetto al suo stato attuale. In altre parole, salviamo la versione corrente del nostro codice in un commit (*processo*). Possiamo creare quanti commits vogliamo e possiamo spostarci avanti e indietro fra commit per vedere le diverse versioni del progetto. Questo ci permette di tenere traccia del nostro progetto man mano si sviluppa e gestire il suo progresso.

Commit sono solitamente creati in punti logici: dopo aver aggiunto un contenuto specifico, una caratteristica o una modifica.

Prima di fare un commit sul nostro codice, dobbiamo aggiungerlo all’interno della **staging area**.

3.1 git add

Dalla cartella del progetto, possiamo usare git add per aggiungere i nostri file nella staging area che permette di tracciarli.



Working directory **Staging Area**

Possiamo aggiungere un file specifico alla staging area con il seguente comando:

```
git add (nome del file)
```

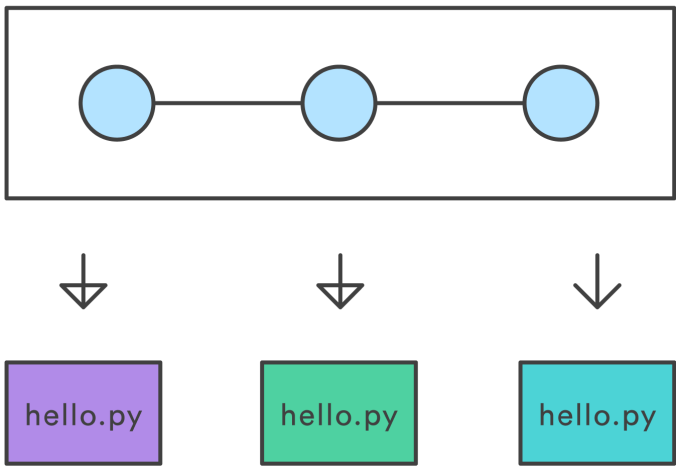
Possiamo anche aggiungere più file.

Invece di aggiungere i file individualmente, possiamo aggiungerli tutti insieme dalla cartella del progetto alla staging area:

```
git add .
```

3.2 git commit

Un commit è uno snapshot del nostro codice in un momento particolare che noi salviamo nella cronologia dei commit del nostro repository. Dopo aver aggiunto tutti i file che vogliamo tracciare alla staging area con il `git add`, siao pronti a fare un commit.



Per eseguire i commit dei file dalla staging area, usiamo il comando seguente:

```
git commit -m "messaggio riguardo il commit"
```

All'interno delle virgolette dovremmo scrivere un messaggio usato per identificarlo nella cronologia dei commit. Questo messaggio dovrebbe descrivere in maniera riassuntiva i cambiamenti effettuati alla repository.

`git commit --amend` è un modo conveniente per modificare il commit più recente. Ti consente di modificare il commit precedente invece di creare un commit completamente nuovo. Può anche essere utilizzato per modificare semplicemente il messaggio del commit precedente senza modificarne lo snapshot, ma, la modifica non solo altera il commit più recente, lo sostituisce
interamente, il che significa che il commit modificato sarà una nuova entità con il proprio ***ref****(puntatore)*.

```
git commit --amend -m "messggio del commit "
```

Per creare un nuovo commit, dovrai ripetere il processo di aggiungere i file alla staging area e fare il commit dopo. È molto utile usare il `git status` per vedere quali file sono

stati modificati.

3.3 git stash

`git stash` archivia temporaneamente le modifiche che hai apportato alla tua copia di lavoro in modo da poter lavorare su qualcos'altro, quindi tornare indietro e applicarle nuovamente in seguito. Lo stashing è utile se devi cambiare rapidamente contesto e lavorare su altro e sei a metà di una modifica del codice o non sei ancora pronto per il commit.

La ***stash*** (*scorta*) è salvata in locale nel tuo repository Git quindi non verrà trasferita al server quando si esegue il push.

```
git stash
```

Puoi riapplicare le modifiche precedentemente nascoste con `git stash pop` :

```
git stash pop
```

Facendo ciò le modifiche nella tua stash verranno cancellate e applicate alla working copy.

Per evitare ciò, usando `git stash apply` , le tue modifiche verranno applicate alla working copy senza rimuoverle dalla stash.

```
git stash apply
```

Non sei limitato a una singola scorta. Puoi eseguire `git stash` più volte per creare più nascondigli, quindi utilizzarli `git stash list` per visualizzarli. Per impostazione predefinita, gli stash sono identificati semplicemente come "WIP" (*working in progress*) in cima al ramo e al commit da cui hai creato lo stash.

```
git stash list
```

Dopo un po' può essere difficile ricordare cosa contiene ogni scorta, quindi per capire meglio l’argomento di ogni stash si scrive un commento che descrive quello che si è

fatto.

```
git stash save "messaggio"
```

Quando si hanno liste di stash, per riapplicare una specifica stash alla working copy si esegue il solito comando `git stash pop` con l'aggiunta di un indice.

```
git stash pop stash@{indice}
```

Di standard, *git stash pop* applica la stash più recente: **stash@{0}**. Come si è potuto intuire, l'indice delle stash va in ordine decrescente, quindi lo stash meno recente sarà quello con l'indice più alto.

3.4 git ignore

Per ignorare file che non vuoi che siano tracciati o aggiunti alla staging area, puoi creare un file chiamato `.gitignore` nella tua cartella principale del progetto.

All'interno del file, puoi fare una lista di tutti i file e cartelle che sicuramente non vuoi tracciare.

Per capire come lavorare con .gitignore utilizza questo sito:

> How to Use a .gitignore File
>
> Adam Garrett-Harris When you make commits in a git repository, you choose which files to stage and commit by using git add FILENAME and then . But what if there are some files that you never want to commit? It's too easy to accidentally commit them (especially if you use to stage all files in the current directory).
>
> https://www.pluralsight.com/guides/how-to-use-gitignore-file

---

4. Ispezione di una repository

4.1 git status

Il `git status` comando viene utilizzato per visualizzare lo stato della directory di lavoro e dell'area di staging.

```
git status
```

Ci fa vedere quali file sono stati cambiati, tracciati, etc.

Possiamo aggiungere i file dei progetti non tracciati alla staging area basandoci sulle informazioni prese dal comando `git status`.

Più tardi, `git status` riporterà le modifiche che abbiamo fatto ai nostri file tracciati prima di decidere se aggiungerli alla staging area di nuovo.

```
lucaprotelli@Air-di-Luca ~ % /Desktop/test git status On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

4.2 git log

Per vedere tutti i commit fatti sul nostro progetto possiamo usare questo comando:

```
git log
```

I registri mostreranno i dettagli per ogni commit, come il nome dell'autore, l'hash generato per il commit, la data e l'orario del commit e il messaggio che abbiamo fornito in precedenza.

Per andare ad uno stato precedente del tuo codice del progetto, puoi usare il seguente comando:

```
git checkout <commit-hash>
```

Cambia `<commit-hash>` con l'hash per il commit specifico che vuoi visitare, il quale è in lista con il comando `git log.`

Per andare all'ultimo commit (la versione più nuova del nostro codice di progetto), puoi scrivere questo comando:

```
git checkout master
```

Questo è un esempio di come esce il registro di un commit:

```
commit b0f0667fe701562a175dca939a256d2e5ccf7e3f (HEAD -> main, origin/main)
Author: lucaprotelli <lucaprotelli@outlook.it>
Date:   Tue Jan 25 21:12:51 2022 +0100

    added main go with empty main function

commit 2475f7a540fbe4715632b20842aff5ba5646e4f1
Author: lucaprotelli <lucaprotelli@outlook.it>
Date:   Tue Jan 25 18:25:16 2022 +0100

    added body{} in main.css

commit c40de45007d4f332aabb3a1d3936a7ca8a5ee64c
Author: lucaprotelli <lucaprotelli@outlook.it>
Date:   Mon Jan 24 23:09:40 2022 +0100

    added console.log

commit 73ff19d85dfdf235ad3f9f41174533428d75230d
Author: lucaprotelli <lucaprotelli@outlook.it>
Date:   Mon Jan 24 22:15:44 2022 +0100

    bootstrap project
```

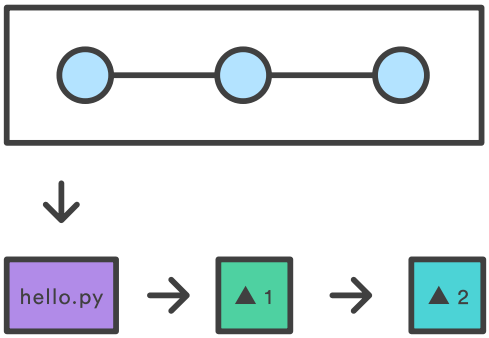
Se stai lavorando ad un progetto grande, il git log risulterà incasinato perché ci saranno molti più commit; un modo per risolvere questo problema è usare `git log —oneline`:

```
01feaa0 (HEAD -> main, origin/main) Update main.go
a618252 Create README.md
b0f0667 added main go with empty main function
2475f7a added body{} in main.css
c40de45 added console.log
73ff19d bootstrap project
```

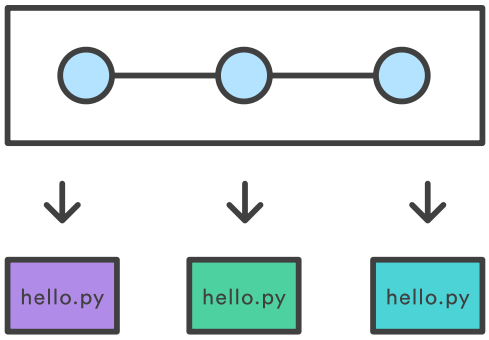
4.3 git status vs git log

Mentre `git status` ti consente di ispezionare la working directory e la staging area, `git log` opera solo sulla cronologia dei commit.

Recording File Diffs (SVN)



Recording Snapshots (Git)



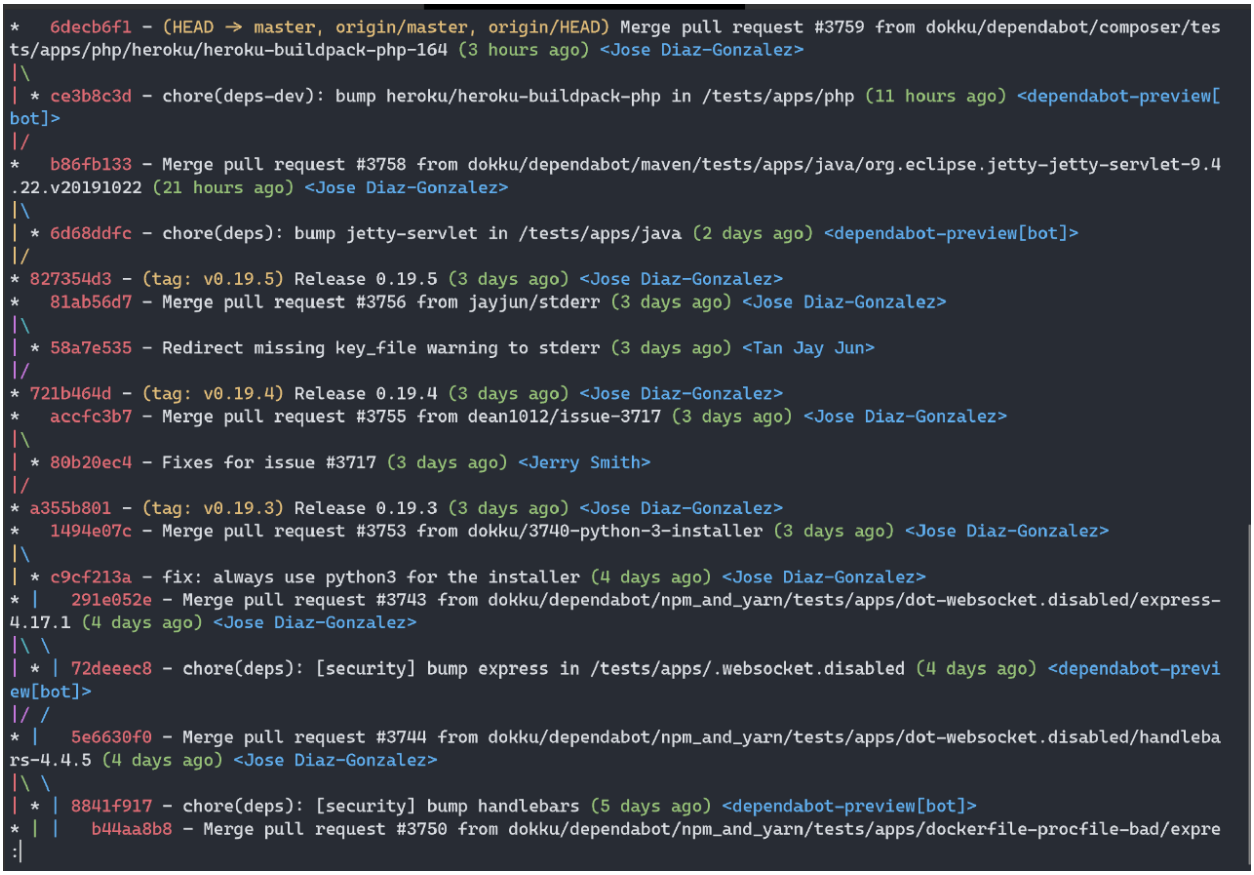
5. Annullamento di commit e modifiche

Per capire bene come funziona l’annullamento in Git, bisogna pensare a esso come ad un'utilità di gestione della sequenza temporale. I commit sono snapshot di un momento o di punti di interesse lungo la timeline della cronologia di un progetto. Inoltre, è possibile gestire più timeline attraverso l'uso di branch. Quando si "annulla" in Git, di solito si torna o indietro o in un'altra timeline in cui non si sono verificati errori.

5.1 Annullamento di commit e modifiche

L'idea alla base di qualsiasi sistema di controllo della versione è archiviare copie "sicure" di un progetto in modo da non doversi mai preoccupare di rompere irreparabilmente la base di codice. Dopo aver creato una cronologia dei commit del progetto, puoi rivedere e rivisitare qualsiasi commit nella cronologia. Per far ciò si utilizza `git log`.

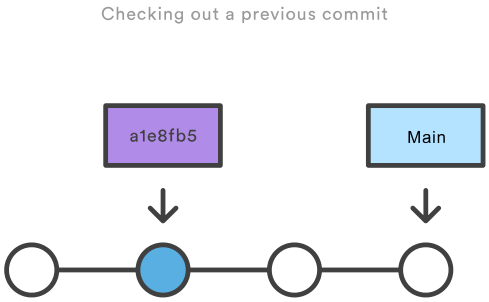
Esempio:



Ogni commit ha un hash di identificazione SHA-1 univoco *(ex: 6decb6f1)*. Questi ID vengono
utilizzati per percorrere la timeline impegnata e rivisitare i commit. Di default, `git log` mostra solo i commit del branch attualmente selezionato; è possibile però che il commit che stai cercando sia su un altro ramo. Puoi visualizzare tutti i commit su tutti i branch eseguendo `git log --branches`. Usando invece `git branch-a` verrà restituito un elenco di tutti i branch conosciuti. Uno di questi, può essere registrato utilizzando `git log`.

Quando hai trovato il riferimento del commit al punto della cronologia(*history*) che desideri visitare, puoi utilizzare il comando `git checkout` per ispezionare il commit. `Git`

`checkout` serve a "caricare" gli snapshots sul pc. Mentre lavoriamo al progetto, l'HEAD di solito punta al main o a qualche altro branch locale, ma quando si estrae un commit precedente, l'HEAD non punta più ad un branch, ma punta direttamente ad un commit.



L'estrazione di un vecchio file non sposta l'HEAD, infatti, rimane sullo stesso branch e sullo stesso commit. Puoi quindi eseguire il commit della vecchia versione del file in un nuovo snapshot come faresti con qualsiasi altra modifica. Quindi questo utilizzo di `git checkout` su un file serve per tornare a una vecchia versione di un singolo file.

Esempio:

```
git log --oneline
872fa7e Ho provato una funzionalità strana
a1e8fb5 Modifica importante al file "prova.txt"
435b61d Creato prova.txt
9773e52 Initial import
```

Questo esempio lo utilizzerò per spiegare la differenza dei comandi che si usano di solito per l'annullamento di commit e modifiche. L'obbiettivo dell'esercizio è quello di annullare il commit **872fa7e** che risulta, in un progetto ipotetico, una modifica sbagliata che vogliamo annullare.

5.2 git rm

Per rimuovere un file dalla staging area, esegui questo comendo

```
git rm --cached
```

5.3 git checkout

Usando il comando `git checkout` possiamo controllare il commit precedente, `a1e8fb5`, mettendo il repository in uno stato prima che si verificasse il commit sbagliato. Il checkout di un commit specifico metterà il repository in uno stato di "detached HEAD". Ciò significa che non lavori più su nessuna branch. In uno stato “detached”, tutti i nuovi commit che esegui saranno resi “orfani” quando cambi branch in un branch stabilito. I commit orfani verranno eliminati dal Garbage Collector di Git. Per evitare che i commit orfani siano nel garbage collector, dobbiamo assicurarci di trovarci su un branch stabilito.

Dallo stato "detached HEAD", possiamo eseguire `git checkout -b nuovo-branch`. Questo creerà un nuovo branch chiamato `nuovo-branch` che passerà a quello stato. Il repository è ora su una nuova timeline della cronologia in cui il commit `872fa7e` non esiste più. A questo punto, possiamo continuare a lavorare su questo nuovo branch in cui il commit `872fa7e` non esiste più e considerarlo “undone”*(annullato)*. Questa strategia di annullamento non è appropriata se il branch in questione si tratta del main; questo perché il main branch non può essere mai cancellato dato che è il ramo portante di tutto il progetto.

5.4 git revert

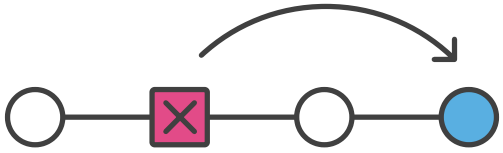
Supponiamo di essere tornati al nostro esempio di cronologia dei commit originale. Questa volta proviamo un annullamento di ripristino. Se eseguiamo `git revert HEAD`, Git creerà un nuovo commit con il contenuto opposto dell'ultimo commit, questo vuol dire annulla tutte le modifiche eseguite nell’ultimo commit. Questo aggiunge un nuovo commit alla cronologia del ramo corrente e ora lo fa sembrare:

```
git log --oneline
e2f9a78 Revert "Ho provato una funzionalità strana"
872fa7e Ho provato una funzionalità strana
a1e8fb5 Modifica importante al file "prova.txt"
435b61d Creato prova.txt
9773e52 Initial import
```

A questo punto, abbiamo di nuovo tecnicamente "undone"*(annullato)* il commit `872fa7e`. Sebbene `872fa7e` esista ancora nella cronologia, il nuovo commit `e2f9a78` è l'inverso delle modifiche in `872fa7e`. A differenza del metodo usando “checkout”, possiamo continuare a utilizzare lo stesso ramo. Questo è il metodo di "annullamento" ideale per

lavorare con repository condivisi pubblici. Questa strategia non è adatta se vuoi avere una cronologia minima e ottimizzata.

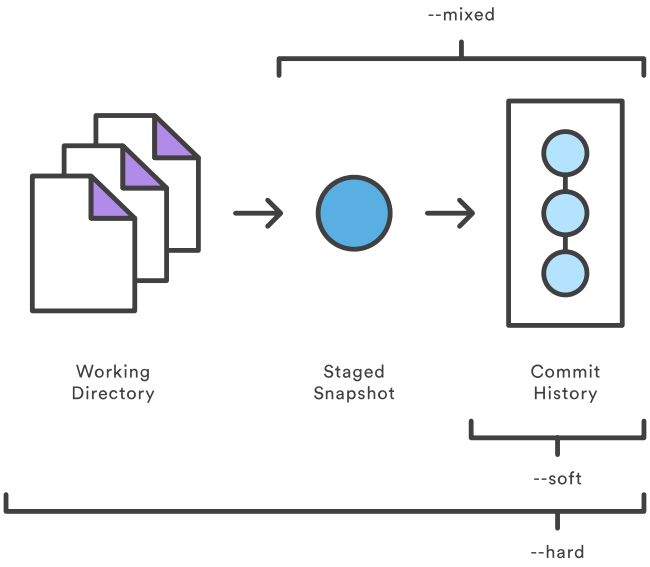
Il ripristino dovrebbe essere utilizzato quando si desidera applicare l'inverso di un commit dalla cronologia del progetto. Questo può essere utile, ad esempio, se stai rintracciando un bug e scopri che è stato introdotto da un singolo commit, invece di entrare manualmente, aggiustarlo ed eseguire un commit per un nuovo snapshot, puoi usare `git revert`.



5.5 git reset

Il `git reset` comando è uno strumento complesso e versatile per annullare le modifiche. Ha tre forme principali di invocazione. Questi moduli corrispondono agli argomenti della riga di comando `-soft, --mixed, --hard`. I tre argomenti corrispondono ciascuno ai tre meccanismi interni di gestione dello stato di Git, The Commit Tree*(Commit History)* (`HEAD`), The Staging Index e The Working Directory.

The scope of git reset's modes



`—- hard`

Questa è l'opzione più diretta, PERICOLOSA e usata di frequente. Utilizzando `-hard` i puntatori di riferimento della commit history vengono aggiornati al commit specificato. Quindi, lo staging index e la working directory vengono reimpostati in modo che il contenuto corrisponda a quello del commit specificato. Qualsiasi modifica in sospeso viene reimpostata in modo che corrisponda allo status della working directory. Ciò significa che qualsiasi lavoro in sospeso nello staging index e nella working directory andrà perso.

`—-mixed`

Lo staging index viene reimpostato sullo status del commit specificato. Tutte le modifiche che sono state annullate dallo staging index vengono spostate nella directory di lavoro.

`—-soft`

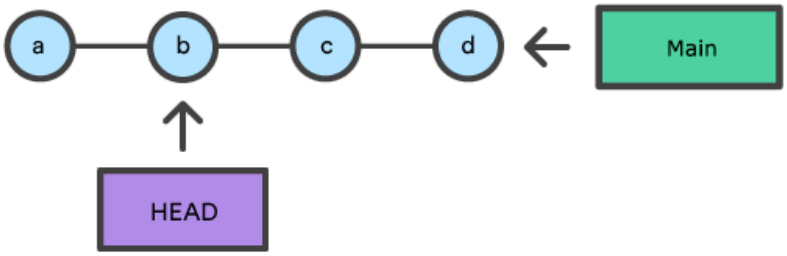
Quando si esegue il reset passando l'argomento `-soft`, i puntatori di riferimento vengono aggiornati e il ripristino si interrompe qui. Lo staging index e working directory non vengono modificati.

Funzionamento:

A livello superficiale, `git reset` ha un comportamento simile a `git checkout`. Laddove `git checkout` opera esclusivamente sull' `HEAD` di riferimento, `git reset` sposterà l'`HEAD` e il branch di riferimento corrente. Per dimostrare meglio questo comportamento, si consideri il seguente esempio:

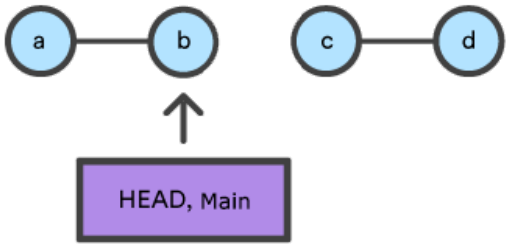
Questo esempio mostra una sequenza di commit sul main branch. L'HEAD e il main branch puntano al commit "d". Ora confrontiamo `git checkout b` e `git reset b.`

git checkout b



Con `git checkout`, il `main ref` sta ancora puntando a `d`. L'HEAD è stato spostato e ora punta al commit `b`. Il repository è ora in uno stato di “detached HEAD”.

git reset b



`git reset` sposta entrambi al commit specificato. Oltre ad aggiornare i puntatori di riferimento nel commit, `git reset` modifica lo stato delle tra aree di lavoro. La modifica del puntatore di riferimento avviene sempre nella Commit History.

Di default, eseguire `git reset` equivale a eseguire `git reset --mixed HEAD`. In questa forma `HEAD` è il commit specificato. Invece di `HEAD` qualsiasi hash di commit Git SHA-1 può essere utilizzato.

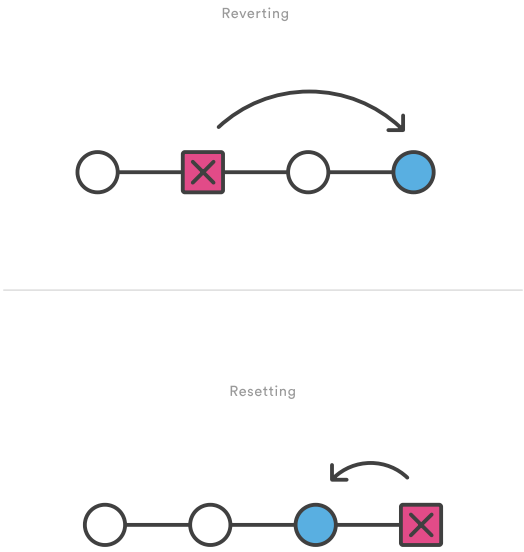
Se invochiamo, `git reset --hard a1e8fb5` la cronologia dei commit viene reimpostata su quel commit specificato. L'esame della cronologia dei commit con `git log` ora sarà simile a:

```
git log --oneline
a1e8fb5 Modifica importante al file "prova.txt"
435b61d Creato prova.txt
9773e52 Initial import
```

L'output del log mostra che i commit `e2f9a78` e `872fa7e` non esistono più nella cronologia dei commit. A questo punto, possiamo continuare a lavorare e creare nuovi commit come se i commit sbagliati non fossero mai avvenuti. Questo metodo di annullamento delle modifiche ha l'effetto più netto sulla cronologia*(history)*. Eseguire un ripristino è ottimo per le modifiche locali, tuttavia aggiunge complicazioni quando si lavora con un repository remoto condiviso. Se abbiamo un repository remoto condiviso a cui è stato inviato il commit `872fa7e` e proviamo ad effettuare un `push` del branch in cui abbiamo ripristinato la cronologia, Git lo catturerà e genererà un errore. Git presumerà che il branch sottoposto a push non sia aggiornato a causa della mancanza di commit. In questi casi, usare `git revert` è la scelta più saggia.

5.6 git revert vs git reset

Mentre il reverting è progettato per annullare in modo sicuro un commit pubblico, `git reset` è progettato per annullare le modifiche locali allo Staging Index e alla Working Directory. Possiamo dire che i due hanno finalità distinte: questo perché il resetting rimuove completamente un set di modifiche, mentre il reverting mantiene il gruppo di modifiche originale e utilizza un nuovo commit per applicare l'annullamento.



Vantaggi del reverting:

- Non cambia la history del progetto = operazione sicura per i commit pubblicati in un repository condiviso
- Capacità di indirizzare un singolo commit in un punto arbitrario della cronologia

6. Riscrivere la Commit History

6.1 git commit —amend

Il comando `git commit --amend` è serve per modificare il commit più recente. Ti consente di combinare le modifiche con il commit precedente invece di creare un commit completamente nuovo. Ma, la modifica non solo altera il commit più recente, lo sostituisce interamente, il che significa che il commit modificato sarà una nuova entità con il proprio puntatore.

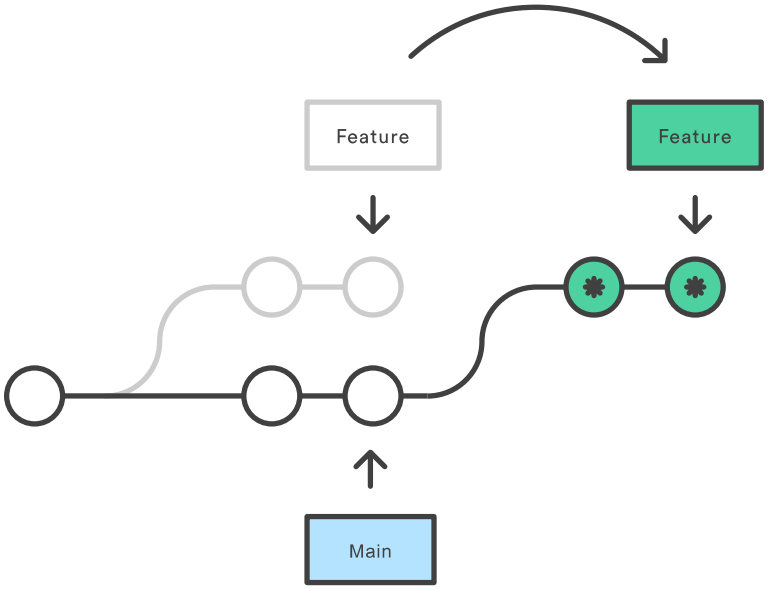
```
git commit --amend (per modificare un commit)
```

Può anche essere utilizzato per modificare semplicemente il messaggio del commit precedente senza modificarne lo snapshot.

```
git commit --amend -m "messaggio nuovo"
```

6.2 git rebase

Il rebase è il processo di spostamento o combinazione di una sequenza di commit in un nuovo commit di base. Il rebase è molto utile in un flusso di lavoro di ramificazione delle funzionalità.



Il rebasing sta cambiando la base del tuo ramo da un commit all'altro facendo sembrare che tu abbia creato il tuo ramo da un commit diverso. Internamente, Git realizza questo creando nuovi commit e applicandoli alla base specificata. È molto importante capire che anche se il ramo ha lo stesso aspetto, è composto da commit completamente nuovi. Il motivo principale del rebase è mantenere una history del progetto lineare.

6.3 git e git rebase interattivo (riassunto)

Git rebase ti dà il potere di modificare la tua cronologia e il rebasing interattivo ti consente di farlo senza lasciare tracce "disordinate". Questo crea la libertà di commettere e correggere errori e perfezionare il tuo lavoro, pur mantenendo una cronologia del progetto pulita e lineare.

```
git rebase -i
```

6.4 git reflog

Git tiene traccia degli aggiornamenti sulla punta dei branch utilizzando un meccanismo chiamato log di riferimento o "reflog". Molti comandi Git accettano un parametro per specificare un riferimento o "ref", che è un puntatore ad un commit.

Esempi comuni includono:

- `git checkout`
- `git reset`
- `git merge`

I reflog tengono traccia di quando i riferimenti Git sono stati aggiornati nel repository locale. Oltre ai reflog deli puntatori die branch, viene mantenuto uno speciale reflog per la scorta di Git. I reflog sono archiviati in una directory nella directory del repository locale.

Puoi ottenere un reflog completo di tutti i riferimenti eseguendo:

```
git reflog show --all
```

Per vedere il reflog per un ramo specifico, passare il nome del ramo a `git reflog show`.

```
.git reflog show "nome del branch"
```

---

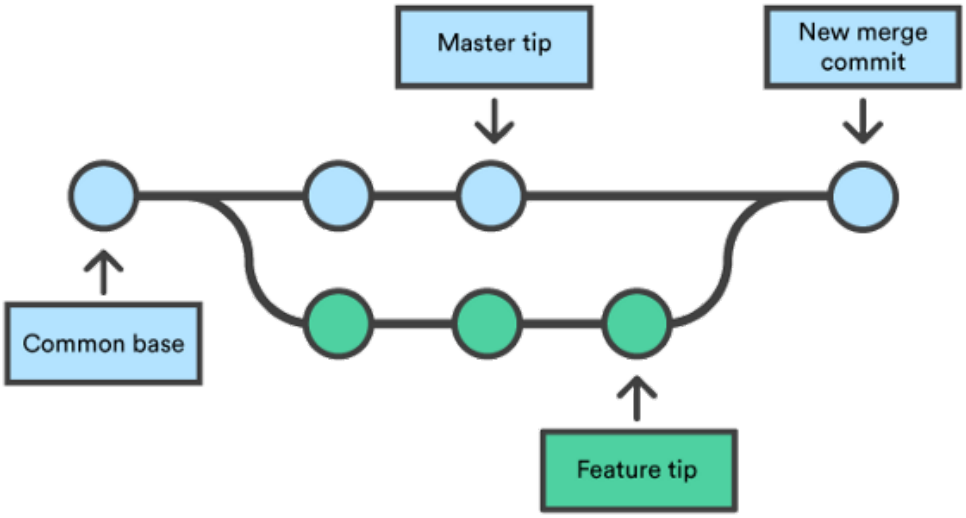
7. I branch in Git

Un ramo (***branch***) potrebbe essere interpretato come una singola timeline dei nostri commits del progetto.

Con Git, possiamo creare molti ambienti (ad esempio, diversi rami) cosicché altre versioni del nostro codice del progetto possano esistere ed essere tracciate in parallelo.

Questo ci permette di aggiungere nuove funzionalità in diversi branch, senza toccare la versione ufficiale del nostro codice del progetto (di solito è tenuto sul master o main branch).

Quando inizializziamo una repository e iniziamo a fare commits, questi sono salvati sul master branch in automatico.



7.1 git branch

Puoi creare un nuovo ramo (***branch***) usando il comando seguente:

```
git branch <nome del branch>
```

Il nuovo ramo (***branch***) che viene creato sarà il referente allo stato attuale del tuo repository.

È una buona idea creare un ramo di sviluppo dove puoi lavorare sul migliorare il codice e aggiungere nuove funzionalità sperimentali. Dopo aver sviluppato e testato le nuove funzionalità per far sì che esse non abbiano bug e che possano essere usate, puoi unirle (merge) al master branch.

Ecco qui alcune piccole funzionalità che potrebbero essere utili:

- Per vedere i branch che hai sulla local repository usa:

  ```
  git branch
  ```

- Per vedere i branch che hai sulla remote repository usa:

```
git branch -r
```

- Per vedere i branch che hai sia sulla local repo e anche sulla remote repo usa:

```
git branch -a
```

- Per rinominare un branch, puoi effettuare il `git brach` con il `-m`:

```
git branch -m "nuovo nome del branch"
```

- Per eliminare un branch, puoi effettuare il `git brach` con il `-d`:

```
git branch -d "il nome del branch che vuoi eliminare"
```

- Se vuoi creare un branch partendo da commit più vecchi rispetto a quello precedente, usa:

```
git branch "nome del branch" codice hash
```

7.2 git checkout

Per spostarsi ad un ramo diverso, usi il `gitcheckout`:

```
git checkout "nome del branch in cui vuoi spostarti"
```

Con quello, ti puoi spostare ad una diversa timeline del tuo progetto cambiando branch.

Per esempio, potresti lavorare su diverse funzionalità nel tuo codice e avere branch separati per ciascuna funzionalità. Quando cambi branch, puoi eseguire i commit sui cambiamenti del codice, i quali influenzano solo quel branch in particolare. In seguito, puoi passare ad un altro branch per lavorare su una funzionalità diversa, la quale non sarà influenzata dai cambiamenti effettuati sul branch precedente.

Ecco qui altre funzionalità che ti possono essere d’aiuto:

- Per ritornare al branch precedente usa questo comando:

```
git checkout -
```

- Per creare un nuovo branch e cambiarlo allo stesso tempo, puoi usare `-b`:

```
git checkout -b "nome del nuovo branch"
```

7.3 git merge

Puoi effettuare il merge dei branch in situazioni in cui vuoi implementare i cambiamenti del codice che hai fatto in un branch individuale ad un altro.

Per esempio, dopo che hai implementato e testato del tutto una nuova funzionalità nel tuo codice, vorrai unire (***merge***) quei cambiamenti al main branch del tuo progetto.

Per effettuare un merge dei cambiamenti da un branch diverso al tuo branch attuale, puoi usare questo comando:

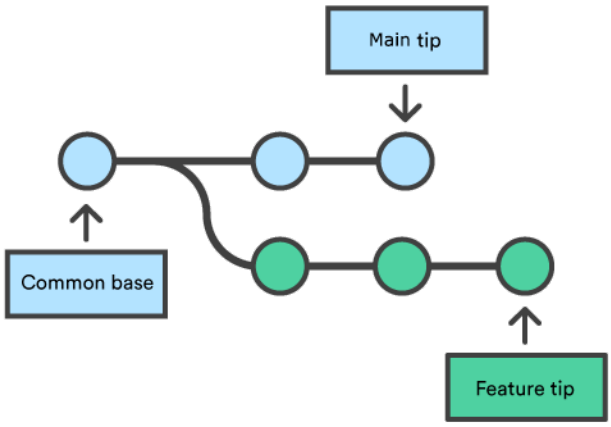
```
git merge "nome del branch"
```

Dovresti cambiare `nome del branch` con il branch che vuoi integrare al tuo branch corrente.

`git merge` viene spesso utilizzato insieme `git checkout` per selezionare il ramo corrente e `git branch -d` per eliminare il ramo di destinazione obsoleto.

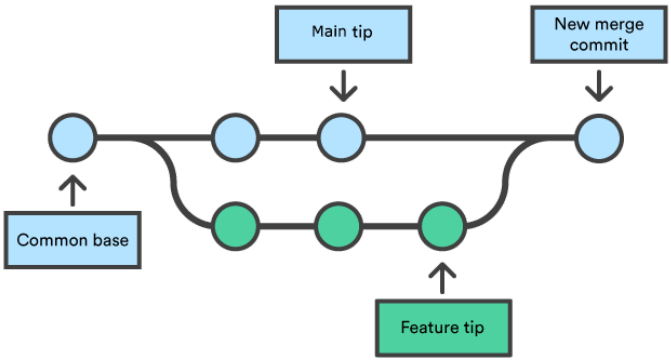
Funzionamento: quando eseguiamo un merge, git prende i puntatori dei due commit per trovare il “base commit”. Una volta trovato, Git creerà un nuovo “merge commit” che combina le modifiche dei due commit da unire.

Esempio: Supponiamo di aver creato un nuovo branch per fare una nuova funzionalità. Una volta finita, vogliamo unire il branch al main.



- Ramo ricevente
- Ramo di fusione *(merging branch)*

Usando il merge, Git unirà il *“Feature tip”* con il *“Main tip”.*



Quando si crea un “merge commit”, Git tenterà di unire automaticamente le history separate. Se Git incontra un dato che viene modificato in entrambe le history, non sarà in grado di combinarli automaticamente. In questo caso si verifica un conflitto di controllo della versione e Git avrà bisogno dell'intervento dell'utente per continuare.

Prima di eseguire un merge bisogna eseguire un paio di controlli per garantire che il merge avvenga senza problemi:

- Esegui `git status` per assicurarti che `HEAD` stia puntando al ramo che riceverà l’unione. Se necessario, eseguire `git checkout` per passare al ramo corretto. Nel

nostro caso eseguiremo `git checkout main`.

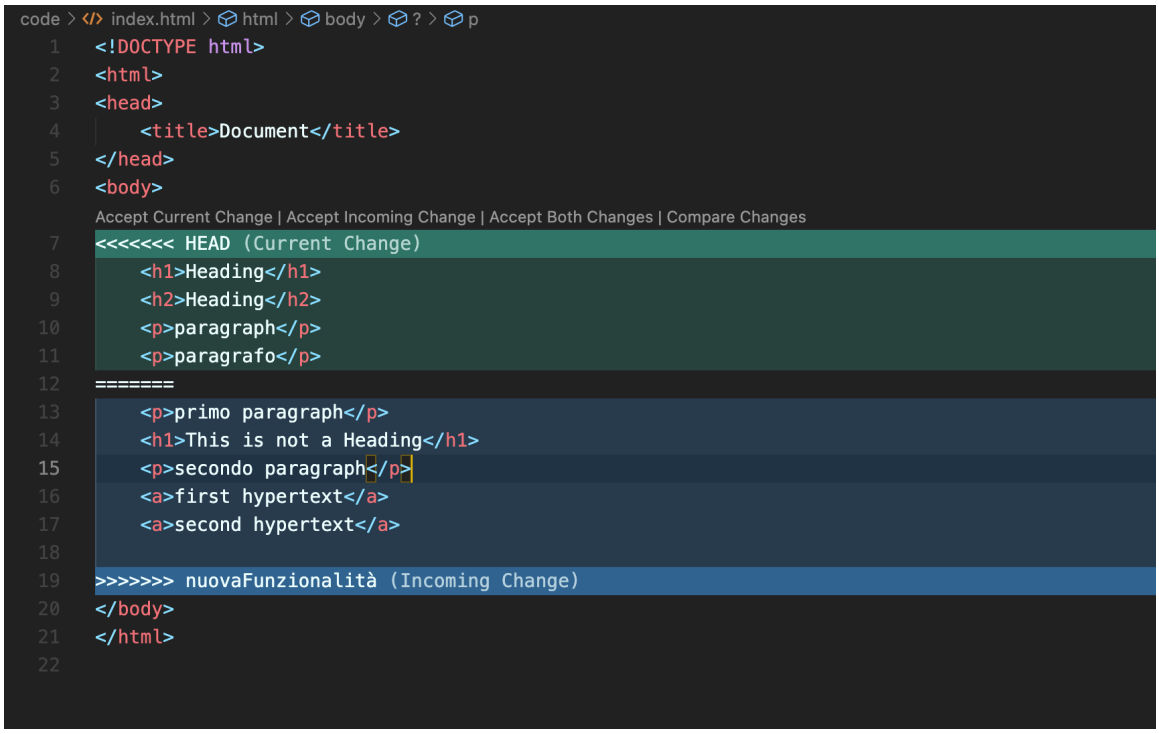
- Assicurati che il ramo ricevente e il ramo di fusione siano aggiornati con le ultime modifiche remote. Esegui `git fetch` per avere gli ultimi commit remoti. Una volta completato il recupero, assicurati che il `main branch` abbia gli ultimi aggiornamenti eseguendo `git pull`.

Come capire se si sono verificati dei conflitti: Git produrrà un output descrittivo che ci farà sapere che si è verificato un CONFLICT. Per vederlo eseguiamo git status. Quando Git incontra un conflitto durante un'unione, modificherà il contenuto dei file interessati con indicatori visivi che contrassegnano entrambi i lati del contenuto in conflitto.

Per vederli bisogna esaminare il file in questione. Questi indicatori visivi sono:

- `<<<<<<<`
  - `=======`
- `>>>>>>>`

Questa linea `======` indica il “centro del conflitto”. Dico così per spiegare il fatto che tutto il contenuto tra il centro e questa linea `<<<<<<` è contenuto che esiste nel ramo ricevente. Viceversa, tutto il contenuto tra il centro e questa linea `>>>>>>>` è contenuto presente nel nostro ramo di fusione.



Come risolvere i conflitti: Dopo aver identificato le sezioni in conflitto, puoi entrare e correggere l'unione a tuo piacimento. Quando sei pronto per completare l'unione, tutto ciò che devi fare è eseguire `git add` sui file in conflitto per dire a Git che sono stati risolti. Dopo, esegui un normale `git commit` per generare il “merge commit”. È esattamente lo stesso processo di un commit di uno snapshot normalissimo, questo vuol dire che è facile per gli sviluppatori gestire i propri merge.

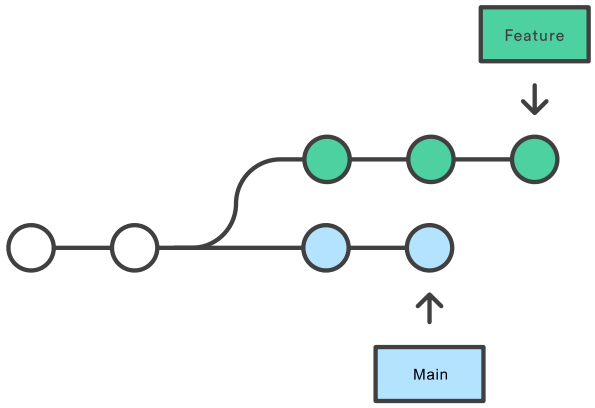
I conflitti *(riassunto):* Git può gestire la maggior parte delle unioni da solo con le funzionalità di fusione automatica. Si verifica un conflitto quando due rami separati hanno apportato modifiche alla stessa riga in un file o quando un file è stato eliminato in un ramo ma modificato nell'altro. È molto probabile che si verifichino conflitti quando si lavora in un ambiente lavorativo perché si è in un numero elevato di contributori.

7.4 git merge vs git rebase

La prima cosa da capire è che `git rebase` risolve lo stesso problema di `git merge`. Entrambi questi comandi sono progettati per integrare le modifiche da un ramo all'altro, semplicemente lo fanno in modi molto diversi. Considera cosa succede quando inizi a

lavorare su una nuova funzionalità in un ramo dedicato, quindi un altro membro del team aggiorna il `main branch` con nuovi commit.

A forked commit history



Ora, supponiamo che i nuovi commit nel `main` siano rilevanti per la funzionalità su cui stai lavorando. Per incorporare i nuovi commit nel tuo `feature branch`, hai due opzioni: merge o rebase.

- **merge**

  L'opzione più semplice consiste nell'unire il `main branch` nel ramo delle `feature` utilizzando:

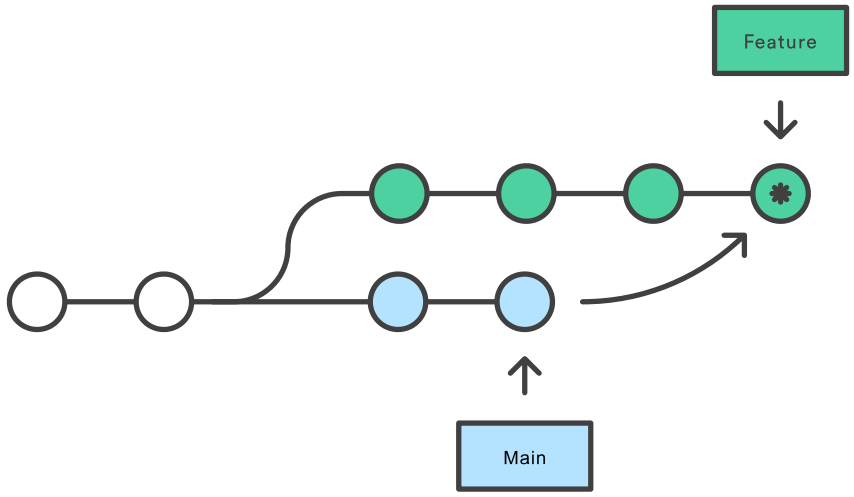
  ```
  git checkout feature
  git merge main
  ```

  Oppure questo in una riga:

  ```
  git merge feature main
  ```

  Questo crea un nuovo "merge commit" nel `feature branch` che unisce le history di entrambi i rami, dandoti una struttura del ramo simile a questa:

Merging main into the feature branch



Il merge è figo perché è un'operazione *non distruttiva* . I branch esistenti non vengono modificati in alcun modo. Ciò evita tutte i potenziali problemi del `ribase`.

D'altra parte, questo significa anche che il `feature branch` avrà un “merge commit” estraneo ogni volta che è necessario incorporare modifiche. Se nel `main` vengono importate molte modifiche, questo può rovinare un po' la history del tuo `branch feature`. Sebbene sia possibile risolvere questo problema con `git log`, può rendere difficile per altri sviluppatori comprendere l’history del tuo lavoro.

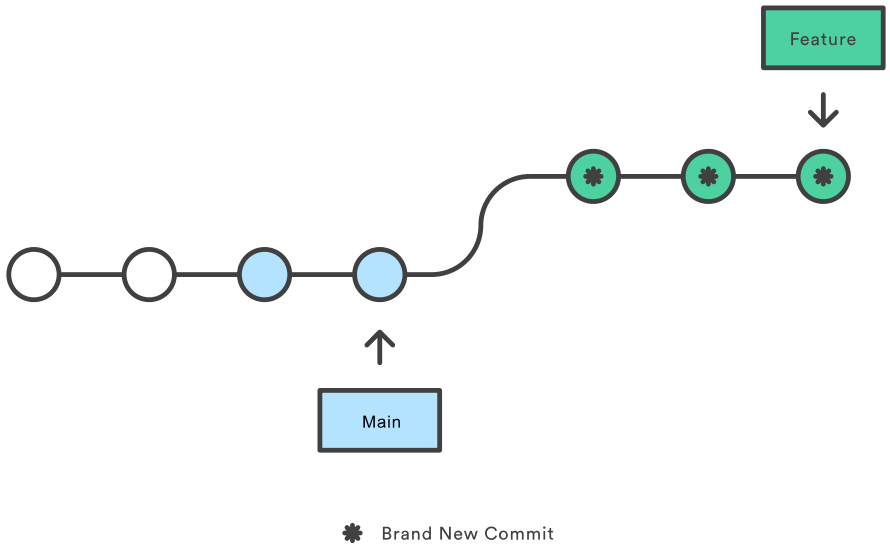
- **rebase**

  In alternativa al merge, puoi fare il rebase del `feature branch` sul `main branch` usando i seguenti comandi:

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

  Questo sposta l'intero `feature branch` sulla punta del `main branch`, incorporando efficacemente tutti i nuovi commit nel `main`. Ma, invece di utilizzare un “merge commit”, il rebasing *riscrive* la cronologia del progetto creando nuovi commit per ogni commit nel `main branch`.

Rebasing the feature branch onto main



Il principale vantaggio del rebase è che ottieni una cronologia del progetto molto più pulita. Innanzitutto, elimina i “merge commit” non necessari richiesti da `git merge`. In secondo luogo, come puoi vedere nel diagramma sopra, il ribase rende la cronologia del progetto perfettamente lineare. Ciò semplifica la navigazione nel progetto ai tuoi colleghi.

Ma ci sono due compromessi per questa storia di commit incontaminata: sicurezza e tracciabilità. Se non segui determinate regole la riscrittura della cronologia del progetto può provocare seri problemi per il tuo workflow di collaborazione. Un’altro problema del rebase è che non puoi vedere quando le modifiche sono state incorporate nel `feature branch`, a differenza del merge che grazie alla creazione di un “merge commit”, hai la possibilità di controllare tutte le volte che hai apportato modifiche dal `main branch` nel `feature branch`.

---

8. La sincronizzazione in Git

Git fornisce ad ogni sviluppatore la propria copia del repository, completa della propria history locale e della struttura dei branch. Gli utenti in genere devono condividere una serie di commit anziché un singolo set di modifiche. Invece di eseguire il commit di una

modifica da una copia di lavoro al repository centrale, Git ti consente di condividere interi rami tra i repository.

8.1 git remote

Il `git remote` comando consente di creare, visualizzare ed eliminare connessioni ad altre repository. Quando si clona una repository con `git clone`, viene creata automaticamente una connessione remota che punta alla repository clonata. Ciò è utile per gli sviluppatori che creano una copia locale di una repository remota, poiché fornisce un modo semplice per eseguire modifiche o pubblicare commit locali.

```
git remote
```

Elenca le connessioni remote che hai ad altri repository.

```
git remote -v
```

Come il comando precedente, ma include l'URL di ogni connessione.

Il comando `git remote` è anche un metodo pratico per modificare il file `./.git/config` di una repository. I comandi presentati di seguito consentono di gestire le connessioni con altri repository. I seguenti comandi modificheranno il file `/.git/config` della repository. Il risultato dei seguenti comandi può essere ottenuto anche modificando direttamente il file `./.git/config` con un editor di testo.

```
git remote add <nome> <url>
```

Crea una nuova connessione a una repository remota. Dopo aver aggiunto la repository remota, puoi usare il `<nome>` come comoda scorciatoia per l' `<url>` in altri comandi di Git.

```
git remote rm <nome>
```

Rimuove la connessione alla repository remota chiamata `<nome>`.

```
git remote rename <vecchio nome> <nuovo nome>
```

Rinomina una connessione remota da `<vecchio nome>` al `<nuovo nome>`.

Git è progettato per offrire a ogni sviluppatore un ambiente di sviluppo completamente isolato. Ciò significa che le informazioni non vengono passate automaticamente tra le repository ma gli sviluppatori devono eseguire manualmente il pull dei commit nella propria repository locale o eseguire manualmente il push dei propri commit nella repository remota.

8.3 git fetch

Il comando `git fetch` scarica commit, file e riferimenti da una repository remota nella repository locale. Git isola il contenuto recuperato dal contenuto locale già esistente; non andrà assolutamente in contrasto/conflitto sul tuo lavoro in locale. Il contenuto recuperato deve essere estratto in modo utilizzando `git checkout`. Questo rende il recupero un modo sicuro per rivedere i commit prima di integrarli con il tuo repository locale.

Applicazione:

Recupera tutti i branch dalla repository. Scarica anche tutti i commit e i file richiesti dall'altra repository.

```
git fetch <remote>
```

Come il comando precedente, ma recupera solo il branch specificato.

```
git fetch <remote> <branch>
```

Recupera tutte le remote repository registrate e le loro branch:

```
git fetch --all
```

8.2 git push

Viene utilizzato per caricare il contenuto della repository locale in una repository remota.

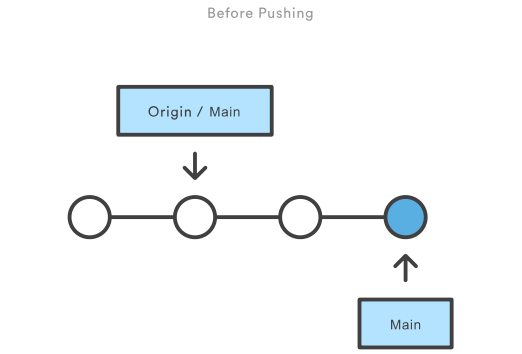
- Il comando seguente invia le modifiche apportate sul master branch.

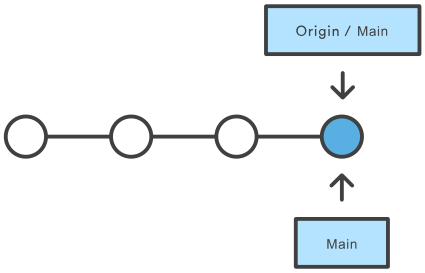
```
git push origin master/master
```

- Il comando seguente invia tutti i branch alla repository remota.

```
git push --all
```

Dopo che un repository locale è stato modificato, viene eseguito un push per condividere le modifiche con i membri del team remoti.





Il diagramma sopra mostra cosa succede quando il tuo local `main` ha superato la repository centrale "`main`" e pubblichi le modifiche eseguendo `git push origin master/main`. Nota come `git push` è essenzialmente lo stesso che eseguire `git merge main` dall'interno del repository remoto.

8.3 git pull

Questo comando viene utilizzato per accedere alle modifiche (commit) da una repository remota alla repository locale. Puoi anche passare un URL.

```
git pull
```

Bibliografia 

https://git-scm.com/doc

https://docs.github.com/en

https://www.atlassian.com/git