GIT CLONE:

comando viene utilizzato per creare una copia di un repository o un branch specifico all'interno di un repository.

SINTASSI → `git clone` https://github.com/github/training-kit.git

`(git clone + link cartella github)`

Quando cloni un repository, non ottieni un file, come potresti fare in altri sistemi di controllo della versione centralizzati. Clonando con Git, ottieni l'intero repository: tutti i file, tutti i rami e tutti i commit.

La clonazione di un repository in genere viene eseguita solo una volta, all'inizio dell'interazione con un progetto. Una volta che un repository esiste già su un telecomando, come su GitHub, dovresti clonare quel repository in modo da poter interagire con esso localmente. Dopo aver clonato un repository, non sarà necessario clonarlo di nuovo per eseguire uno sviluppo regolare.

La possibilità di lavorare con l'intero repository significa che tutti gli sviluppatori possono lavorare più liberamente. Senza essere limitato dai file su cui puoi lavorare, puoi lavorare su un feature branch per apportare modifiche in sicurezza. Allora puoi:

- utilizzare successivamente `git push` per condividere il branch con il repository remoto;
- apri una pull request per confrontare le modifiche con i tuoi collaboratori;
- testare e distribuire secondo necessità dal branch;
- fondersi nel branch `MAIN`;

GIT STATUS:

mostra lo stato corrente della tua directory di lavoro Git e dell'area di staging.

Permette di vedere quali modifiche sono state sottoposte a staging.

Nel dubbio, run `git status`. Questa è sempre una buona idea. Il comando `git status` genera solo informazioni, non modificherà i commit o le modifiche nel tuo repository locale.
Una caratteristica utile di `git status` è che fornirà informazioni utili a seconda della situazione attuale. In generale, puoi contare su di esso per dirti:

- Dove `HEAD` sta puntando, che si tratti di un branch o di un commit (questo è il punto in cui sei "estratto");
- Se nella directory corrente sono presenti file modificati che non sono ancora stati sottoposti a commit;
- Se i file modificati vengono messi in scena o meno;
- Se il tuo attuale ramo locale è collegato a un branch remoto, allora `git status` ti dirà se il tuo branch locale è in ritardo o in anticipo rispetto a qualsiasi commit;

Durante i conflitti di unione, `git status` ti dirà anche esattamente quali file sono la fonte del conflitto.

GIT ADD:

comando aggiunge file nuovi o modificati nella tua directory di lavoro all'area di staging di Git.

`git add` è un comando importante - senza di esso nessuno `git commit` farebbe mai niente. A volte, `git add` può avere la reputazione di essere un passaggio non necessario nello sviluppo. Ma in realtà, `git add` è uno strumento importante e potente. `git add` ti permette di plasmare la storia senza cambiare il modo in cui lavori.

Quando lo usi? Mentre lavori, modifichi e salvi un file o più file. Quindi, prima di committare, devi fare `git add.` Questo passaggio ti consente di scegliere cosa committare. I commit dovrebbero essere unità di cambiamento logiche e atomiche, ma non tutti funzionano in questo modo. Forse stai apportando modifiche a file che *non sono* unità di modifica logiche o atomiche. `git add` ti consente comunque di modellare sistematicamente i tuoi impegni e la tua cronologia.

Cosa fa? `git add [filename]` seleziona quel file e lo sposta nell'area di staging, contrassegnandolo per l'inclusione nel commit successivo. È possibile selezionare tutti i file, una directory, file specifici o persino parti specifiche di un file per lo staging e il commit.
Ciò significa che se fai un `git add` in un file eliminato, l'eliminazione viene messa in scena per il commit. Il linguaggio di "aggiungi" quando stai effettivamente "eliminando" può creare confusione. Se pensi o usi `git stage` al posto di `git add`, la realtà di ciò che sta accadendo potrebbe essere più chiara.
`git add` e `git commit` vanno insieme mano nella mano. Non funzionano quando non vengono utilizzati insieme. Ed entrambi funzionano meglio se usati pensando alla loro funzionalità articolare.

GIT COMMIT:

crea un commit, che è come un'istantanea del tuo repository, più precisamente manda cose nella repository locale, creando una storia. Questi commit sono istantanee dell'intero repository in momenti specifici. Dovresti fare spesso nuovi commit, basati su unità logiche di cambiamento. Nel tempo, i commit dovrebbero raccontare una storia della storia del tuo repository e di come è diventato quello che è attualmente. I commit includono molti metadati oltre ai contenuti e al messaggio, come l'autore, il timestamp e altro.

Come funziona? I commit sono gli elementi costitutivi dei "punti di salvataggio" all'interno del controllo della versione di Git.

Sintassi:

```
git commit -m "aggiorna il README.md con il link alla guida per contribuire"
```

Commit cronologia della forma

Usando i commit, sei in grado di creare la cronologia intenzionalmente e in sicurezza.

Puoi effettuare commit su branches diversi e specificare esattamente quali modifiche desideri includere. I commit vengono creati sul branch in cui sei attualmente in check-out (ovunque punti HEAD), quindi è sempre una buona idea eseguire git status prima di effettuare un commit, per verificare di aver effettuato il check-out nel branch che intendi essere. Prima di eseguire il commit, dovrai mettere in scena tutte le nuove modifiche che desideri includere nel commit utilizzando git add [file].

I commit sono hash SHA leggeri, oggetti all'interno di Git. Finché lavori con file di testo, non dovrai preoccuparti di quanti file hai, quanto sono grandi o quanti impegni fai. Git può gestirlo!

Commit in due fasi

I commit hanno due fasi per aiutarti a creare correttamente i commit. I commit dovrebbero essere unità di cambiamento logiche e atomiche che rappresentano un'idea specifica. Ma non tutti gli esseri umani lavorano in questo modo. Potresti lasciarti trasportare e finire per risolvere due o tre problemi prima di ricordarti di impegnarti! Va bene - Git può gestirlo. Una volta che sei pronto per creare i tuoi commit, utilizzerai `git add <FILENAME>` per specificare i file che desideri "mettere in scena" per il commit. Senza aggiungere alcun file, il comando `git commit` non funzionerà. Git guarda solo all'area di staging per scoprire cosa impegnare. Lo staging o l'aggiunta di file è possibile tramite la riga di comando e anche con la maggior parte delle interfacce Git come GitHub Desktop selezionando le righe o i file che si desidera mettere in scena.
Puoi anche utilizzare un comodo comando, `git add -p`, per esaminare le modifiche e separarle, anche se si trovano nello stesso file.

GIT PUSH:

carica tutti i commit del branch locale nel ramo remoto corrispondente.

`git push` aggiorna il branch remoto con commit locali. È uno dei quattro comandi in Git che richiede l'interazione con il repository remoto. Puoi anche pensare `git push` ad *aggiornare* o *pubblicare*.
Per impostazione predefinita, `git push` aggiorna solo il branch corrispondente sul telecomando. Quindi, se si esegue il check-out al `main` branch quando si esegue `git push`, verrà aggiornato solo il `Main` branch. È sempre una buona idea usare `git status` per vedere su quale ramo ti trovi prima di passare al telecomando.
Dopo aver apportato e confermato le modifiche in locale, puoi condividerle con il repository remoto utilizzando `git push`. L'invio di modifiche al telecomando rende i tuoi impegni accessibili ad altri con cui potresti collaborare. Questo aggiornerà anche tutte le richieste pull aperte con il branch su cui stai lavorando.
Come best practice, è importante eseguire il comando `git pull` prima di inviare eventuali nuove modifiche al branch remoto. Questo aggiornerà la tua filiale locale con eventuali nuove modifiche che potrebbero essere state inviate al remoto da altri contributori. Il pull prima del push può ridurre la quantità di conflitti di unione che crei su GitHub, consentendoti di risolverli localmente prima di inviare le modifiche al branch remoto.

GIT PULL:

aggiorna il tuo attuale branch di lavoro locale e tutti i branches di tracciamento remoto. È una buona idea eseguire `git pull` regolarmente nei branches su cui stai lavorando a livello locale.
Senza `git pull` (o l'effetto di esso) la tua filiale locale non avrebbe nessuno degli aggiornamenti presenti nel canale di Github.
`git pull` è una delle 4 operazioni remote all'interno di Git. Senza eseguire `git pull`, il tuo repository locale non verrà mai aggiornato con le modifiche dal remoto. `git pull` dovrebbe essere usato ogni volta che interagisci con un repository remota, come minimo. Ecco perché `git pull` è uno dei comandi Git più utilizzati.

`git pull`, una combinazione di `git fetch` + `git merge`, aggiorna alcune parti del tuo repository locale con modifiche dal repository remoto. Per capire cosa è e cosa non è influenzato da `git pull`, devi prima capire il concetto di branches di tracciamento remoto. Quando si clona un repository, si clona un branch di lavoro, `main` e tutti i rami di tracciamento remoto. `git fetch` aggiorna i branches di tracciamento remoto. `git merge` aggiornerà il tuo branch corrente con eventuali nuovi commit sul branch di tracciamento remoto.

`git pull` è il modo più comune per aggiornare il tuo repository.

Tuttavia, potresti voler utilizzare `git fetch` invece. Uno dei motivi per farlo potrebbe essere che ti aspetti dei conflitti. I conflitti possono verificarsi in questo modo se si hanno nuovi commit locali e nuovi commit in remoto. Proprio come un conflitto di unione che si verificherebbe tra due diversi branches, queste due diverse righe della cronologia potrebbero contenere modifiche alle stesse parti dello stesso file. Se si opera per la prima volta `git fetch`, l'unione non verrà avviata e non verrà richiesto di risolvere il conflitto. Questo ti dà la flessibilità di risolvere il conflitto in un secondo momento senza la necessità di connettività di rete.

Un altro motivo per cui potresti voler eseguire `git fetch` è l'aggiornamento a tutti i branches di tracciamento remoto prima di perdere la connettività di rete. Se esegui `git fetch`, e successivamente provi a eseguire `git pull` senza alcuna connettività di rete, la `git fetch` parte dell'operazione `git pull` avrà esito negativo.
Se usi `git fetch` invece di `git pull`, assicurati di ricordarti di `git merge`. L'unione del branch di tracciamento remoto nel tuo branch garantisce che lavorerai con eventuali aggiornamenti o modifiche.

GIT LOG:

Mostra i log dei commit, quindi Elenca i commit raggiungibili seguendo i `parent` collegamenti dai commit dati, ma escludi i commit raggiungibili da quelli dati con un ^ davanti. L'output viene fornito in ordine cronologico inverso per impostazione predefinita.

Puoi pensare a questo come a un'operazione impostata. I commit raggiungibili da un qualsiasi dei commit dati sulla riga di comando formano un insieme, quindi i commit raggiungibili da uno qualsiasi di quelli dati con ^ davanti vengono sottratti da tale insieme. I restanti commit sono ciò che esce nell'output del comando. Varie altre opzioni e parametri di percorso possono essere utilizzati per limitare ulteriormente il risultato.

In poche parole mostra la storia della directory o del file interessato.

GIT DIFF:

Mostra le modifiche tra commit, commit e albero di lavoro, ecc.

Quindi mostra le modifiche effettuate confrontando la storia della directory o del file interessato.

GIT CHECKOUT:

Cambia branches o ripristina i file dell'albero di lavoro.

Aggiorna i file nell'albero di lavoro in modo che corrispondano alla versione nell'indice o nell'albero specificato. Se non è stato fornito pathpec, `git checkout` aggiornerà anche `HEAD` per impostare il branch specificato come branch corrente.

GIT REVERT:

Ripristina alcuni commit esistenti.

Dati uno o più commit esistenti, ripristinare le modifiche introdotte dalle relative patch e registrare alcuni nuovi commit che li registrano. Ciò richiede che il tuo albero di lavoro sia pulito (nessuna modifica dal commit HEAD).

Nota: `git revert` viene utilizzato per registrare alcuni nuovi commit per invertire l'effetto di alcuni commit precedenti (spesso solo uno difettoso).

Bisogna specificare il commit che voglio tornare. In poche parole questo comando va nel passato e cerca di applicare le modifiche correnti, non cancellando la storia.