QUESTO È UN FILE SU GIT

Disclaimer: l'autore di questo file non garantisce la totale comprensione.

In questo file cercherò di rispondere a tutte le domande su Git, spiegando le sue funzioni.

1. **Che cos'è GIT?**
   Git è uno strumento che ci permette di collaborare e scambiare progetti, tenendo traccia dei cambiamenti apportati. È meglio usare Git piuttosto che avere 10 file diversi chiamati "Tesi finale", "Tesi finale 2", "Tesi finale 2: la vendetta di Joker", ecc. Ognuno di questi apporta una modifica (che non ci è data sapere a colpo d'occhio) ed occupano memoria inutile nei nostri dispositivi.
2. **Come si usa?**
   Utilizzando Git Bash (consigliato) o qualsiasi command line (purchè si abbia il file eseguibile di Git sul computer)
3. **Come posso muovere i primi passi?**
   Utilizza il comando `git help` per visualizzare i vari comandi che si possono eseguire con una piccola spiegazione annessa per ciascuno di loro
4. **Creazione di un "piano di lavoro" (DIRECTORY)**
   Cominciamo con il creare una semplice cartella nella quale lavoreremo. Eseguiamo i seguenti passaggi:
   a. `cd Desktop` ---------- decidiamo di creare la nostra cartella sul desktop
   b. `mkdir Miadirectory` ---------- creazione della cartella
   c. `cd Miadirectory` ---------- entriamo nella cartella così da poterci lavorare
   d. `git init` ---------- inizializziamola, ovvero rendiamola una REPOSITORY (o Repo)
5. **Cos'è una Repo?**
   Una Repo altro non è che il contenitore ufficiale dei nostri file, la versione definitiva dentro la quale mettiamo i file che consideriamo buoni, validi.
6. **Ma cosa stiamo facendo?**
   Attualmente abbiamo creato una cartella "vuota". In realtà dentro troviamo una cartella nascosta creata automaticamente da Git che non dovremo toccare: serve al programma per tenere traccia dei cambiamenti. Per sapere in che stato ci troviamo eseguiamo il comando `git status`. In questo modo riusciamo ad avere una visione chiara del nostro lavoro svolto fino ad ora.
7. **Come posso interpretare questo stato?**
   Immaginiamoci di essere diventati improvvisamente Dante Alighieri e vogliamo intraprendere il viaggio raccontato ne "La Divina Commedia". Come tutti sappiamo Dante attraversa 3 mondi prima di uscire "*a riveder le stelle*": Inferno, Purgatorio e Paradiso. Analogamente, in Git abbiamo: Local, Staging Area e Repository. Consideriamo l'Inferno come lo stato nel quale abbiamo creato dei file che non abbiamo ufficializzato o non abbiamo messo nella "Staging Area", il nostro Local. Qui noi abbiamo apportato delle modifiche al file senza renderle note a Git (in realtà lui riconosce che ci sono state delle modifiche ma non tiene ancora traccia di esse).
8. **Vediamo come salvare un file con le sue modifiche**
   Creiamo un file di testo all'interno della nostra cartella e chiamiamolo come vogliamo, ad esempio "curriculum", e scriviamoci dentro il nostro nome. A questo punto vogliamo salvare questa creazione del file. Se eseguiamo di nuovo il comando `git status` noteremo che avremo un file non tracciato (proprio il nostro curriculum). A questo punto aggiungiamolo nella "Staging Area", il Purgatorio, tramite il comando `git add` curriculum (occasionalmente aggiungere .txt oppure premere il tasto TAB dopo aver scritto qualche lettera del nome). Eseguendo nuovamente `git status`, notiamo che abbiamo un file "ready to be committed".

9. **Cos'è un commit?**
   Quando vogliamo salvare un file con i relativi cambiamenti nella Repo significa che vogliamo "committare", ovvero ufficializzare il nostro lavoro mettendolo nel Paradiso. È fortemente consigliato aggiungere un messaggio a questo commit, così una volta che guarderemo tutti i commit fatti, sapremo a colpo d'occhio cosa abbiamo fatto ad ogni cambiamento. Per committare con messaggio (in questo caso il nostro curriculum) useremo il comando `git commit -m "creazione curriculum"`. Se ora rivedessimo lo status, Git ci direbbe "nothing to commit".
10. **Modifichiamo il file**
    Come detto precedentemente, Git è utile per modificare i file e tenere traccia delle modifiche. Mettiamo caso che siamo stupidi e abbiamo scritto male il nostro nome, vogliamo sicuramente cambiarlo. Scriviamo il nome corretto e ripetiamo i passaggi precedenti (`git add` e `git commit`).
11. **Che cosa abbiamo fatto finora?**
    Adesso abbiamo fatto solamente due commit, quindi a meno di attacchi di Alzheimer cronici dovremmo ricordarci tutto, però in lavori ben più grandi è utile vedere cosa abbiamo fatto. Usando il comando `git log` vedremo la lista di tutti i commit fatti, con i relativi messaggi e codici hash identificativi.
12. **Muoviamoci tra i branch**
    Attualmente stiamo lavorando sul branch **master** (o **main**). Immaginiamoci questo branch come una linea temporale scandita dai vari commit. Potrebbe capitare di dover lavorare su alcuni file ma non vogliamo creare casino nella branch principale. Bisogna creare un'altra branch così il nostro piano di lavoro da lontano sembrerà un albero. Creiamola con il comando `git branch test`: questa altro non sarà che una copia della main branch. Per spostarci su quella branch eseguiamo il comando `git checkout test` (esiste anche il comando per spostarsi su una branch e crearla nel caso in cui non esistesse: `git checkout -b test`).
    Adesso che abbiamo una nuova branch di test dove lavorare possiamo fare quello che vogliamo senza modificare ciò che abbiamo fatto sulla main branch. A questo punto nel nostro file "curriculum" aggiungiamo un indirizzo e committiamo questa modifica con il relativo messaggio comunicativo. Ora ci ritroviamo con la main branch "indietro di un passo". Se facciamo un checkout sulla main branch e controlliamo i vari commit fatti (ricordiamo i comandi checkout e log) notiamo che l'aggiunta dell'indirizzo non esiste. Benissimo, abbiamo lavorato sulla test branch e abbiamo deciso che tutto il lavoro svolto può essere implementato nella main branch: è il momento di "mergiare". Rimanendo sulla main branch digitiamo "`git merge test`": così facendo la main branch implementa tutti gli step AGGIUNTIVI (quindi quelli mancanti) della test branch. N.B. per fare il merge, la branch sulla quale ci troviamo deve essere indietro con i passaggi, ovvero non è possibile mergiare una branch che si trova indietro perché entreremmo in un conflitto: Git non saprebbe quale dei due cambiamenti prendere come ufficiali. Possiamo vedere quali commit sono stati fatti tutte le branch (a prescindere da quale branch abbiamo sotto mano) con il comando `git log --all`.
    N.B. Con Merge prendiamo in blocco i cambiamenti, per prendere specifici cambiamenti esiste il cherry-picking che però non abbiamo ancora visualizzato.
13. **Cosa succede se il lavoro svolto non va bene?**
    Chiaramente può capitare che si lavora su un qualcosa ma ci si rende conto che quello ottenuto non sia un buon risultato. Abbiamo due opzioni a disposizione:
    a. **Git reset <u>Soft</u>** → se eseguiamo un reset soft vuol dire che vogliamo tornare indietro ma non vogliamo perdere ciò che abbiamo fatto (i commit svolti). In questo caso torneremo a qualsiasi commit si voglia e dal quale ripartire, senza eliminare definitivamente i commit "saltati". Ipotizziamo di avere questa situazione:
       A-B-C

Ci troviamo su C come commit ma vogliamo tornare a B. Tornando indietro su B, la HEAD (dove ci troviamo attualmente) di Git si sposterà ma i cambiamenti apportati da C verranno visualizzati nella Staging Area. Pertanto, se volessimo riprenderli, ci basterà committarli.

b. **Git reset <u>Hard</u>** → con questo tipo di reset torniamo al commit voluto perdendo tutto ciò che si lascia per strada.

In ogni caso, il dopo il comando relativo ai due reset (`git reset --soft` e `git reset --hard`) andrà scritto il codice hash del commit dove vogliamo spostarci (che troveremo nella lista mostrata dal comando `git log`).

14. Usiamo GitHub per scambiare file

Git è uno strumento molto utile per lavorare in gruppo, infatti possiamo tutti accedere ai file dei nostri collaboratori (purchè li abbiamo inseriti in una repository online ed averci dato l'accesso). Vediamo come fare:

a. Creiamo un account su GitHub
b. Creiamo una repository online
c. Su Git Bash impostiamo la stessa mail usata per GitHub con il comando `git config --global user.email "miamail@blabla.it"`
d. Dobbiamo creare un punto di accesso, un segnalibro, che ci permetta di comunicare con questa repository online (per convenzione lo chiamiamo origin, ma si può chiamare come si vuole). `git remote add origin **link della repository**`. Il link si può copiare direttamente da GitHub oppure si mette l'URL e si aggiunge `.git`

A questo punto ci troviamo con due repo, una locale ed una online, e vogliamo caricare la nostra locale online: `git push origin main`. Cosa stiamo facendo? Stiamo "pushando" tutto il nostro lavoro online sul segnalibro "origin" nella branch "main". Noi ora possiamo continuare a lavorare sulla locale, sicuri che il lavoro è stato salvato. Attenzione!! Il lavoro non viene salvato in tempo reale: quando decidiamo che i nostri commit possono essere visualizzati da tutti, facciamo il push sulla repo online.

15. Come posso prendere i file da un mio collaboratore e lavorarci sopra?

Così come gli altri possono farsi i cazzi nostri, noi possiamo fare lo stesso con i loro. Come?

e. Cambiamo la nostra directory tornando indietro (`cd ..` – così facendo ci troviamo sul desktop).
f. Andiamo su GitHub e copiamo il link della repository interessata
g. Su Git Bash digitiamo `git clone **link**`
h. Cambiamo la directory con la cartella appena creata nel desktop

Ora possiamo modificare i file della cartella come se fosse nostra, quindi possiamo aggiungere file nella staging area, committare, creare altre branch, ecc. Per apportare le modifiche alla repo online eseguiamo gli stessi step fatti per la nostra. Ovviamente dovremo mettere il nome del segnalibro e della branch corrispondenti a quelli del nostro collaboratore. N.B. Se abbiamo creato un'altra branch dobbiamo pushare anch'essa su GitHub, altrimenti avremo pushato solo una mentre l'altra rimarrà in locale.