Guida a Git

Fonte: https://www.html.it/pag/52699/il-funzionamento-di-git/

- Ma ad ogni modifica devo fare ‘add’ ??

Git è un sistema di controllo di versione distribuito (Distributed Version Control Systems o DVCS).
Ciascun client fa anche da server per se stesso e possiede una copia locale del repository.
Ogni volta che l’utente effettua un commit o salva lo stato del proprio progetto, crea un’immagine di tutti i file presenti in quel momento, salvando un riferimento.
Se alcuni file non sono stati modificati, GIT non li clona ma crea un collegamento agli stessi file della versione precedente.
L’utente può lavorare sulla propria copia locale del repository e rendere pubbliche le modifiche quando il server torna online.

Controllo di versione locale -> solo su proprio pc
Controllo di versione centralizzato -> tutto passa da un server, ma criticità perchè se va giù blocca tutti
Controllo di versione distribuito -> (Git) ogni client può fare da server

i dati vengono rappresentati invece come delle istantanee, o snapshots, quando si memorizza lo stato di un progetto il sistema crea quindi un’immagine di tutti i file al momento corrente; in sostanza Git non fa riferimento alle differenze tra le varie versioni, ma ne “fotografa” gli stati di avanzamento.
Così, nel caso in cui un file non dovesse subire delle modifiche, il DVCS non provvederà a salvarlo nuovamente ma si limiterà a definire un collegamento con il salvataggio già effettuato.

Sezioni:
Directory di Git -> L’area per il salvataggio del database e dei metadati di un progetto, cioè le informazioni copiate in fase di clonazione di un repository
Directory di lavoro -> La copia di backup (checkout) di una determinata versione di un progetto, per ottenerla si estraggono i file dall’archivio della directory di Git in modo da poterli manipolare dal proprio disco.
Area di stage -> Definita anche come “Indice”, è un’area rappresentata da un file contenente le informazioni relative alla commit successiva.
Stati:
committati -> I file sono stati memorizzati localmente nel database
modificati -> I file hanno subito dei cambiamenti ma non sono stati ancora committati.
in stage -> I file sono stati coinvolti da modifiche e verranno inclusi in un’istantanea con la prossima commit.

area di stage -> commit -> directory Git -> checkout -> directory di lavoro -> staging -> area di stage

Tutti i file sono presenti nella directory di lavoro. Se tracciati, vanno a finire anche nell’aria di stage e sarà aggiunti al prossimo commit. Una volta committati, vanno a finire in un’istantanea.

Ci sono 3 diversi file di configurazione, ognuno con priorità sugli altri in modo da cambiare una configurazione in locale da quella generale usata per tutti.

Iscrizione utente -> inserisco delle variabili globali
    git config --global user.name 'quattromori'
    git config --global user.email quattromori@latuaemail.com
Se tolgo --global, farò un override delle informazioni senza modificare le variabili globali

Lista impostazioni -> git config --list
Editor di default -> git config --global core.editor NotePad++

Creare un nuovo progetto Git:
Significa trasformare la directory che ospita il nostro codice in un repository per il versioning. Nel gergo di Git questa operazione si chiama import.
Entra nella cartella che contiene il progetto e digita
    git init
Verrà creata una cartella '.git'. Adesso bisogna registrare i file del progetto preesistenti
    git add nome_file
Sarà così aggiornato l'indice dei contenuti correnti. Per committere invece
    git commit -m 'mio_messaggio_descrittivo'

Note:
- proprietà del commit → https://git-scm.com/docs/git-commit

Cartella '.git'

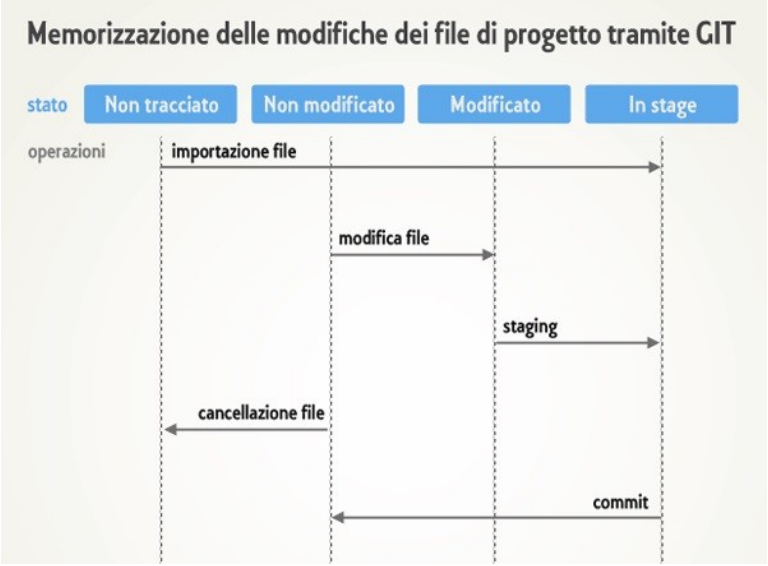
Risorsa	Descrizione
hooks	E' la cartella destinata a contenere script personalizzati (*hooks* o "ganci") per Git, siano essi client-side o server-side, che entreranno in azione in corrispondenza di determinati comportamenti.
info	E' una directory che nativamente presenta il file `exclude` per i pattern delle risorse che non verranno tracciate durante il *versioning*, fa riferimento al file `.gitignore` con il quale escludere estensioni e formati specifici dal tracciamento.
objects	La cartella nella quale vengono archiviati i contenuti per il database.
refs	E' la directory in cui reperire i puntatori per gli oggetti dei commit.
HEAD	Il file che fa riferimento al ramo di sviluppo del quale è stato eseguito il checkout.
description	File non necessario se non si utilizza la piattarfoma GitWeb.
config	Il file che presenta le configurazioni per un determinato progetto.
index	File che memorizza le informazioni relative allo *staging*.

Clonazione:
E' possibile clonare la repository di un progetto con il comando
    git clone URL_progetto_da_clonare
Ricorda di usare Https

Note:

- se un file non viene aggiunto (add) dallo sviluppatore, il programmi non traccerà il file, che risulterà untracked.

Status:
Ci mostra le differenze rispetto al commit precedente, come la presenza di nuovi file non aggiunti e quindi non tracciati. Quindi mostra ciò che comporterà il prossimo commit.



Una volta committato, un file torna ad essere non modificato!
Il branch è il ramo su cui si sta operando. Ci possono essere più rami.

Staging:
entità intermedia fra directory di lavoro e la directory DVCS. Rappresenta tutti i file che riguardano il prossimo commit. Se una modifica di un file non causa l'aggiornamento dell'indice di staging, quella modifica non sarà considerata nel commit.

Se io modifico un file tracciato, devo lanciare 'git add' per metterlo in staging. Ma se io, dopo aver messo un file in staging, rimodifico il file prima di committarlo, esso sarà contemporaneamente in stato modificato e in stage! Devo fare 'git add' per ogni modifica.

E' possibile sapere quali sono le modifiche apportate con il comando
    git diff nome_file_da_controllare
Le modifiche riguardano quelle non ancora in stage, se presenti.

Per bypassare il passaggio nell'area di stage, e quindi non far segnalare che questi file non sono passati senza lo stage, si deve aggiungere al commit l'opzione '-a'.

Gestione dei file

Alcuni file non è necessario inglobarli nel versionamento del progetto, come i log o i file temporanei. Si possono creare delle liste di esclusione dette '.gitignore'. Esse hanno effetto su tutte le directory che non hanno una loro lista di esclusione.

```
Cat .gitignore        # creo una lista
*.log                 # tutti i file con questa estensione
tmp/**/*              #  ignora tutte le sottocartelle e i file di tmp
```

Ci sono molte diciture e regole possibili (https://www.html.it/pag/54414/operazioni-sui-file-con-git/).

Annullamenti

Per l'eliminazione di un file si usa il comando:

```
git rm nome_file
```

che si occuperà di eliminare il file dall'area di stage, committare e infine eliminare fisicamente il file dall'area di lavoro.
Se invece si vuole solo togliere un file dall'area di stage e tenere il file nell'area di lavoro si può usare:

```
git rm –cached nome_file
```

Se invece si elimina a mano un file, esso sarà recepito come modificato ma non aggiornato. Quindi bisognerà comunque lanciare il comando 'git rm'.

E' possibile anche postare o rinominare un file con il comando:

```
mv nome_file nuova_destinazione/nuovo_nome_file
```

Dopo il commit, questa modifica sarà aggiunta alla directory di git.

E' possibile annullare l'ultimo commit, perché si è dimenticato un file o la descrizione è errata, lanciando: git commit –amend

E' possibile annullare una modifica ad un file, riportando il file al commit precedente, ma questo annullamento non sarà rintracciabile perché non rientra in nessun commit: git checkout – nome_file

Unstaging consiste nell'annullare un comando 'git add', cioè nel rimuovere un file nell'ara di stage:

```
git reset HEAD nome_file
```

Log

il comando 'git log' presenta varie opzioni:
- -p → differenze fra i vari commit
- numero → di log da mostrare

Repository remoti

git remote -v→ lista dei server associati al progetto. Più link perché uno per l'invio (push) e uno per la ricezione (fetch).

Per aggiungere un nuovo repository bisogna specificare un nome e il link (tipo github)

```
git remote add nickname link
```

Posso leggere un repository:

```
git fetch nickname
```

Mentre per aggiornare (merging) il mio contenuto con la versione più aggiornata del server:

```
git pull
```

Posso inviare i miei dati sul server:

```
git push nickname nome_del_ramo
```

Per avere varie informazioni su un repository:

```
git remote show nickname
```

Per eliminare la risorsa remota (ma non quelle locali):

```
git remote rm nickname
```

Tag e alias

Per facilitare la ricerca i primi e per semplificare i comandi i secondi (do un mio nome ad un comando esistente).
Si taggano degli stati di avanzamento del progetto, tipo le fasi più importanti:

```
git tag
```

Con l'opzione '-l' seguita dal numero di versione (es. 3*) possiamo specificare quale versione analizzare. Per aggiungere un tag:

```
git tag -a 3.2.2 -m "annotazione"
```

Per avere informazioni su un tag:

```
git tag show versione_da_controllare
```

Si possono condividere i tag con i server remoti:

```
git push origin mia_versione
```

Sarà necessario autenticarsi. Oppure con l'opzione '--tags' saranno condivisi tutti i tag.
Per la rimozione di un tag bisognerà invece usare l'opzione '-d' in locale e puntare al riferimento remoto del tag.

Branch – rami

Per implementare funzionalità isolate ma che partono dalla stessa radice. Il ramo di default è detto master. Posso poi fare il merge con il master quando un ramo si conclude.
Un esempio è quando il ramo principale raggiunge un livello stabile tipo 1.0. Allora si decide di aprire un secondo ramo che rappresenta il 2.0, senza eliminare o stravolgere il ramo principale, che verrà così aggiornato solo quando serve.

Creazione di un ramo:
    git branch nome_ramo
Spostamento su un ramo:
    git checkout nome_ramo
Posso fare entrambe le azioni:
    git checkout -b nome_ramo

Se effettuo ora un commit, questo riguarderà solo il ramo in cui mi sono spostato!
Potrebbe succede che io non posso spostarmi in un ramo (checkout) perché ci sono dei conflitti tra i due rami.
Per fare il merge, ci spostiamo sul ramo che deve rimanere e digitiamo:
    git merge nome_altro_ramo
L'altro ramo si dovrò poi cancellare se non più utile:
    git branch -d ramo_da_cancellare
Per cancellare un ramo che non ha subito fusioni, si utilizza '-D'. E' una questione di sicurezza.
Per avere la lista dei rami basta usare:
    git branch
Usando poi l'opzione '--merged' si può avere la lista dei rami che hanno avuto una fusione, mentre con l'opzione '--no-merge' i rami che non hanno subito fusione.
Per avere informazioni riguardo ai commit di tutti i rami:
    git branch -v

Al posto del merging, si può utilizzare 'rebase'. Funzionano allo stesso modo, fondendo due rami, ma gestiscono la cronologia dei commit in modo differente. Con il rebase si ha una lettura più lineare dei commint derivanti dai due rami che spesso vanno fusi. Va però usato soltanto su rami a cui si lavora da soli.
Es: ramo principale, creo un ramo secondario per aggiungere una funzionalità. Intanto il ramo principale viene aggiornato con altri commit. Io fondo il ramo principale con il ramo secondario varie volte. Se uso il rebase, i commit saranno ben leggibili. Ma se ci lavorasse anche qualcun altro, lui fonderebbe ogni tanto il proprio ramo principale per aggiornare il ramo secondario e non si capirà più nulla. I rami principali di due persone diverse hanno commit diversi!

E' possibile unire un ramo ad un master, specificando di importare soltanto le modifiche non presenti in un terzo ramo:
    git rebase –onto ramo_master ramo_da_non_unire ramo_da_unire
Va poi effettuato anche il merge fra ramo master e quello unito per far in modo che la cronologia coincida. Quando sarà il momento farò la stessa operazione con il terzo ramo e infine cancellerò i due rami perché il master è allineato con entrambi.

Corso udemy

Working directory → contiene i file del nostro progetto
Staging area → è un file dentro la cartella '.git' ed è una zona di transizione che rappresenta un'istantanea di tutte le modifiche (add) che verranno prese in considerazione nel prossimo commit.
Local repository → qui vengono salvate (commit) le istantanee prese dalla staging area.

Il commit crea un oggetto che presenta un nome in SHA1, il riferimento ai propri genitori e l'insieme dei file. Esso è raggiungibile mediante un'intestazione (HEAD). La prima si chiamerà 'Master'.

Dopo aver installato Git, aver aggiunto l'utenza (sia email che nome obbligatori) ed essere entrati nella cartella del progetto, si lancia 'init' perché si crei la cartella nascosta '.git' (visualizza con 'ls -lA').

Per aggiungere tutti i file al commit si usa 'add *'.
Fai il commit con un messaggio di descrizione poi controlla lo storico con 'git log'.

Per fare un commit con un add, cioè tutto insieme, basta usare '-am'

Per tornare ad un commit precedentemente (navigare fra versioni):

```
git checkout ultimi_caratteri_versione_ da caricare
```

In questo modo l'head cambia (?).

Invece di tornare indietro di alcuni commit perché un collega ha fatto danni, è preferibile lavorare su più rami (branch) differenti. Sia per sviluppi indipendenti che per collaborazioni contemporanee.

```
Git branch nuovo_ramo
gir checkout nome_ramo_su_cui_spostarci
```

Il nuovo ramo contiene tutti i commit di quello precedente. IL merge permette di fondere più rami, che poi possono essere cancellati. Se due rami da fondere presentano delle incongruenze, git non ti permette di fondere finché non risolvi le incongruenze. Si entrerà in un branch speciale formato da entrambi i rami e verrà modificato il codice accanto alle incongruenze che ha trovato.

Regole:
- crea un utenza su git dopo l'installazione
- posizionati nella cartella del progetto e inizializza per creare la cartella '.git'
- metti in staging tutti i file da tracciare
- quando serve torna indietro o crea un nuovo ramo
- se lavori con altri prima scarichi la loro versione con fetch, poi la mergi con la tua, fai le tue modifiche, commit e poi push perché anche gli altri ricevano le modifiche che hai fatto.
- incongruenza soltanto nei punti modificati da entrambi gli utenti

Note:
- guida ufficiale → https://git-scm.com/book/it/
- git lavora su molti file quindi limitare il controllo su quelli necessari
- descrizione commit: visibili sono i primi 72 caratteri, che quindi sono i più importanti

Nella cartella .git/objects sono presenti tutti gli oggetti, come i commit. E' possibile ottenere informazioni sul singolo oggetto col comando:

```
git cat-file -p codice_oggetto
```

Nel file HEAD è indicato il riferimento del ramo su cui stai lavorando.
E' possibile lavorando su un repository remoto, come github o bitbucket, ma anche selezionando una cartella del proprio pc. Per creare un repository remoto, dopo aver inizializzato una cartella come progetto vuoto, usare:

```
git remote add origin path_cartella_remota
```

Poi bisognerà esportare i dati con 'push'. Gli altri utenti o quando tu vuoi aggiungere al tuo repository le modifiche apportate da altri, dovranno prima lanciare 'fetch' che crea come un altro ramo. Bisognerà poi fare un merge e risolvere eventuali problemi. Adesso ho aggiornato il mio progetto con le modifiche apportate dagli altri. Alla fine lanciano 'push' per caricare in remoto le mie ultime modifche.
Per controllare quale repository remoto sto usando:

```
git remote -v
```

GIT NIMAIA

Il master di solito indica la produzione. Quindi stare attenti perché poi viene pubblicato in automatico. Gli stati principali di un file sono: nuovo, modificato, cancellato, cambio permessi esecuzione, ecc. Li vedi con ‘git status’.

‘git add .’ o ‘git add --all’ → aggiungo tutti i file (piccola differenza)

‘git restore --stage’ → per staggare e togliere dallo stage

‘git diff’ → confronto con l’ultimo commit (head). Specificando il parametro (head o nome_ramo) fa la differenza con ultimo commit e non con stagin (?testa)

‘gitk –all’ → gui

‘gitk nome_file’ → tutta la storia di quel file

‘git log’ → mostrare lista di commit. Con parametro ‘—grep’ si attiva un filtro per la ricerca

‘nome_ramo’ → (es. master) indica sia un ramo che l’ultimo commit del ramo. Quindi head e master coincidono quando il ramo corrente è master e punto all’ultimo commit.

fork → divergenza di due rami

bare → repository che presenta soltanto i file della cartella .git ma non quelli di progetto (git init –bare)

upstream → relazione fra un ramo remoto e uno locale (non sempre i nomi corrispondo oppure voglio spostare il ramo master nel mio ramo chiamato test). Con ‘clone’ non c’è ne bisogno. Solo ‘pull’ può dare problemi.

revisione → è il codice ahs1 che identifica un commit o la sua etichetta

Remote:
- ‘git remote -v’ → elenco dettagli dei remote attivi
- con ‘clone’ vengono aggiungi i remote in automatico
- ‘git remote add origin mia_path/link → per aggiungere un remote di nome origin
- ‘fetch’ → allineare solo i metadati con il repository remoto e vedere le differenze. Con la proprietà ‘all’ indico tutti i rami.
- ‘git push’ → più proprietà ‘nome_remote’ e ‘nome_ramo’ → invio delle modifiche
- ‘git pull’ → fetch + merge, cioè aggiorno il mio repository locale con i dati presenti in remoto

Commit:
- ‘--amed’ → modifica dell’ultimo commit. Se non ci sono nuovi file, usa ‘-m’ per aggiornare solo il messaggio, altrimenti sostituisce il precedente commit e puoi usare ‘--no-edit’ per lasciare il vecchio messaggio. Non usare se il precedente commit è condiviso con qualcuno”
- **‘reset HEAD+tilde**’ → per tornare al commit precedente. Per spostarmi di più commit usare la ‘tilde’
- ‘git show nome_revision’ → mostra differenze fra 2 commit

- ‘reverse’ → rapido rollback delle modifiche. Crea un nuovo commit che presenta l’inverso delle modifiche recepite da git. Come se tornassi indietro. Se lo rilancio, creo un nuovo commit uguale a quello di partenza !
- ‘git checkout nome_revision -b nome_branch’ → creo un nuovo ramo da un commit vecchio
- .gitignore → file da mettere nella cartella da non caricare. Contiene le regole dei file da escludere. Con ‘*’ indico tutto, mentre con ‘!.gitignore’ dico di non ignorare questo file.

Merge:
- è un nuovo commit che fonde due commit esistenti di rami diversi.
- C’e ne sono di due tipi: fast foward e foward. Il fast si ha quando non ci sono folk e i commit sono consecutivi uno all’altro. Se non è fast, git richiede un commit per allineare i due rami.
- ‘rebase’ → alternativa al merge, aiuta nella lettura dei commit perché va a modificare i metadati di git. E’ come se prima si effettuasse un pull per aggiornare il repository locale e poi aver effettuato la tua modifica. In realtà tu hai fatto una modifica su un codice non aggiornato.

Ridurre numero commit:
- ‘rebase’ → manipolo metadati commit
- ‘reset’ → soft (resetto head ma lascio modifiche) o hard (cancella commit)
- ‘git merge nome_branch –squash’ → considero tutte le modifiche, cioè tutti i commit, del ramo da mergare, come una sola da unire all’ultimo commit dell’altro ramo.

‘Stash’ → mette da parte i file modificati ma non committati, facendoti tornare all’ultimo commit. Così puoi cambiare branch senza dover fare un commit con solo quelle modifiche parziali.
Con ‘git stash list’ ho la lista dei salvataggi fatti. Con ‘git stash apply stash@{0}’ recupero questi file.

Tornare indietro per un solo file: git checkout id_commit_buono – nome_file_da_ripristinare

Note:
- quando cambio di ramo ma ho delle modifiche in corso, devo stare attento perché me le porto sul nuovo ramo. Infatti l’area di stagin non è legata ad un ramo ma è comune!
- .gitkeep → file vuoto da mettere nelle cartelle vuote che vuoi far vedere a git

Configurazioni:
- (Massimo) → mostrare nome branch sul terminale

```
# Show current git branch in command line
parse_git_branch() {
git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\[\033[32m\]$(whoami)@$(hostname):\w\[\033[33m\]\$(parse_git_branch)\[\
033[00m\] $ "
```

- (massimo) → cambiare due comportamenti di git (adesso uso rebase)

```
git config --global push.default upstream

git config --global --bool pull.rebase true
```