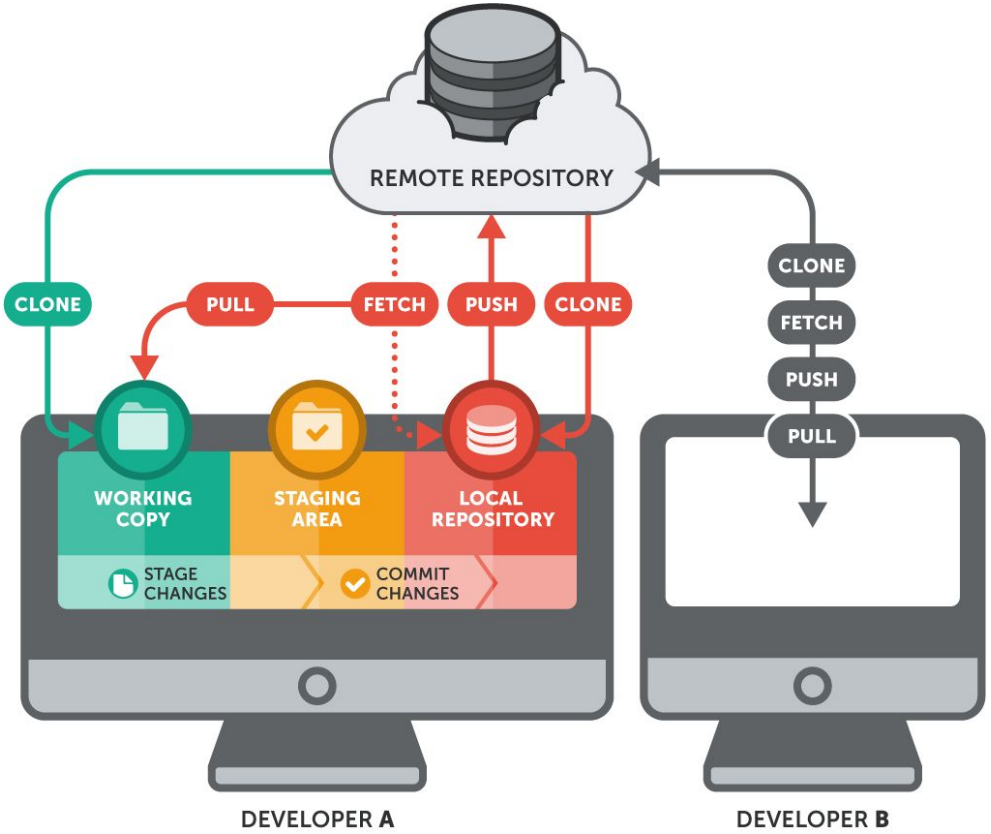
Il controllo di versione con Git



Controllo di Versione

Il **controllo di versione** è un sistema che registra i cambiamenti subiti nel tempo da un file o da una serie di file, così da poter richiamare una specifica versione in un secondo momento.

Un Sistema per il Controllo di Versione (Version Control System – **VCS**) permette di:

- ripristinare i file o l'intero progetto ad una versione precedente,
- revisionare le modifiche fatte nel tempo,
- vedere chi ha introdotto un problema e quando.

Usare un VCS significa anche che se fai un pasticcio o perdi qualche file, puoi facilmente recuperare la situazione.

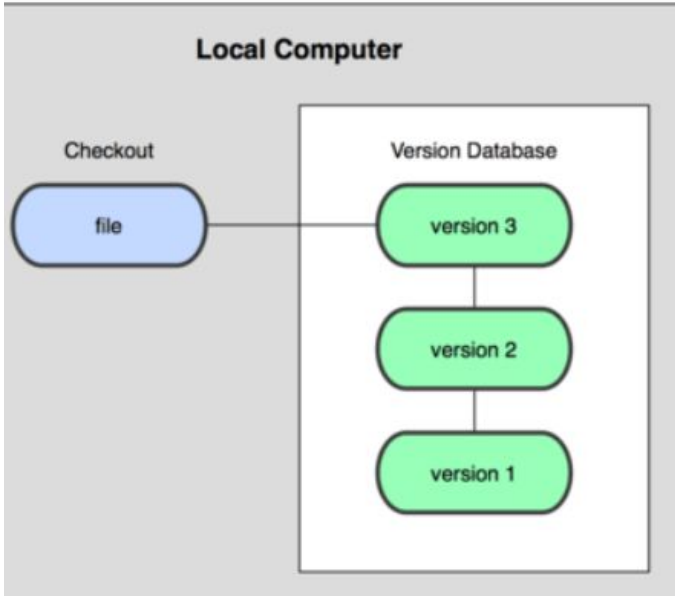
Sistema di Controllo di Versione Locale

All'inizio si adottarono **soluzioni fatte in casa** che prevedevano di gestire le diverse versioni copiando i file in un'altra directory in genere denominata con la data

=> porta a tanti errori, ad esempio è facile dimenticare in quale directory si è, modificare il file sbagliato o copiare i file sbagliati.

PRIMA SOLUZIONE VCS:

I programmatori svilupparono **VCS locali** dotati di un database semplice che manteneva tutti i cambiamenti dei file sotto controllo di revisione

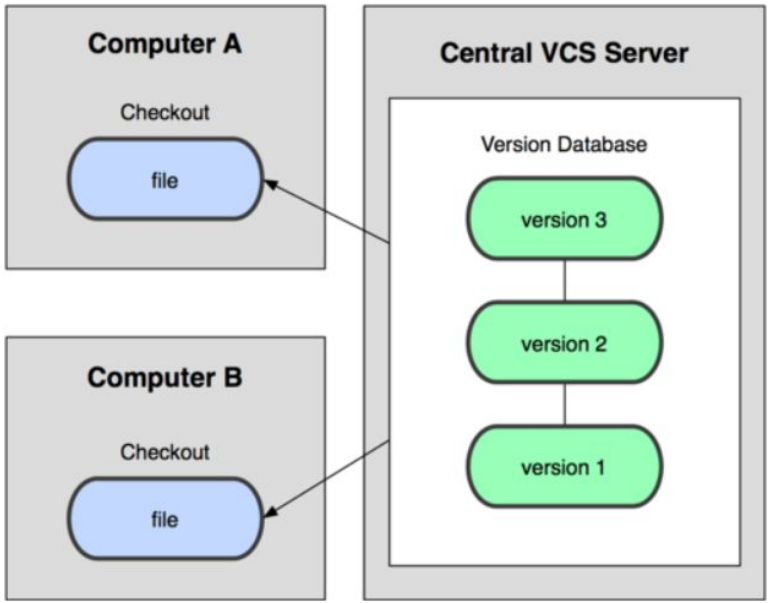


Sistema di Controllo di Versione Centralizzato

Problema: collaborare con altri sviluppatori sullo stesso programma ma su sistemi distinti

SECONDA SOLUZIONE CVCS:

Si svilupparono sistemi di controllo di versione centralizzati (Centralized Version Control Systems – **CVCS**), dove vi era un unico server che conteneva tutte le versioni dei file e i vari utenti scaricavano i file dal server centrale.



- **PRO:** rispetto ai database locali, gli amministratori hanno un controllo preciso su chi può fare cosa, e chiunque vede, con una certa approssimazione, cosa sta facendo un'altra persona del progetto.

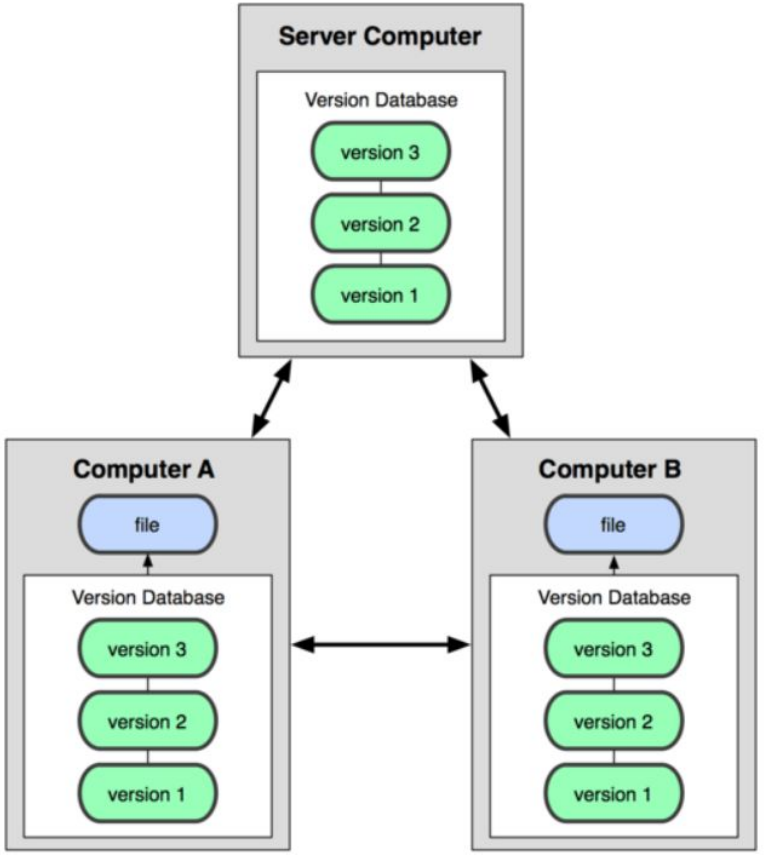
- **CONT:**: ogni volta che tutta la storia del progetto è in un unico posto, si rischia di perdere tutto se qualcosa va storto.

Sistema di Controllo di Versione Distribuiti

TERZA SOLUZIONE DVCS:

in un **DVCS** (come Git, Bazaar), i client non solo controllano lo snapshot più recente dei file, ma fanno una copia completa del repository. In questo modo se un server si danneggia e i sistemi interagiscono tramite il DVCS, il repository di un qualsiasi client può essere copiato sul server per ripristinarlo.

Ogni checkout è un backup completo di tutti i dati.



GIT

- In quanto Distributed Version Control System lo scopo di Git è quello di **permettere la gestione delle modifiche** apportate a una collezione di informazioni, che si tratti del codice sorgente di un'applicazione, della pagina HTML di un sito statico o di documentazione

- La particolarità di Git risiede nel supportare e garantire:
  1. **lo sviluppo distribuito** – ogni developer che partecipa a un progetto non è legato a un server centrale che mantiene la copia "originale" del progetto
  2. **performance elevate per grandi progetti** – lavorare su progetti con una lunga storia o progetti composti da moltissimi file è rapido come lavorare su progetti più giovani o con pochi file
  3. **la salvaguardia della storia delle modifiche** – l'ultima versione è data dalla sequenza esatta di tutte le modifiche apportate nel tempo, pur essendo possibile inserire una nuova modifica tra due modifiche già apportate in passato

tratto da https://aulab.it/guide/62/5-comandi-git-per-sviluppatori-singoli

REPOSITORY GIT LOCALE

Un repository contiene **i file del progetto e la cronologia** delle loro revisioni

- **.git è** la directory nascosta che contiene Il repository. Essa è una sottodirectory della directory principale del proprio progetto locale, ad **es. "my-project"**
- È importante notare che:
  - eliminando la directory **.git** => viene eliminata l'intera cronologia del progetto. La directory my-project torna ad essere una directory "normale"
  - non è detto che tutti i file presenti nella directory my-project facciano parte del repository (solo i file che sono stati espressamente aggiunti al repository entrano di diritto nella history)
- Ci sono due comandi per creare un repository Git locale :
  1. **git init** inizializza il repository in una directory che non è sotto controllo di versione
  2. **git clone** clona un repository esistente

1 Creazione di un repository in una directory esistente

Se si ha una directory di progetto che attualmente non è sotto il controllo della versione e si vuole iniziare a controllarla con Git:

1. entrare nella directory di quel progetto
2. digitare il comando **$ git init**

→ Viene creata nella directory una sotto-directory nascosta **.git** in cui verranno salvati di volta in volta tutti i file necessari a Git per conoscere la cronologia delle modifiche(history) e lo stato attuale.

A questo punto, non è stato ancora tracciato nulla nel tuo progetto

2 Clonazione di un repository GIT esistente

Per ottenere una **copia di un repository GitHub remoto**, ad esempio **https://example.com/project.git**, ed apportare il proprio contributo, si deve eseguire il comando:

$ git clone https://example.com/project.git

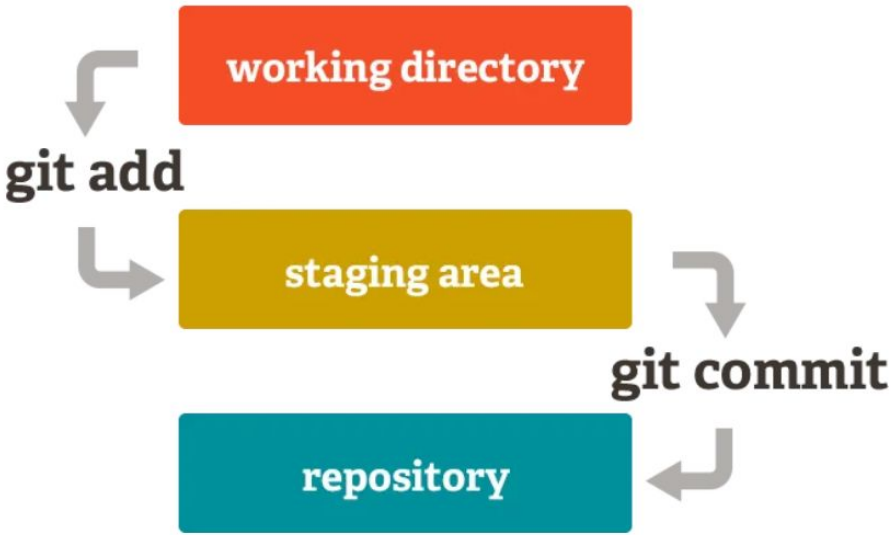
Git compie le seguenti azioni:

- crea una directory locale vuota **project**
- copia nella directory **project/.git** tutti gli snapshot (commit) presenti sul repository remoto)
- individua l'ultimo snapshot (commit) della history
- estrae il contenuto di tutti i file corrispondenti all'ultimo snapshot (commit)

E' possibile cambiare il nome della **directory destinazione** in **progetto1** con il comando:

$ git clone https://example.com/project.git **progetto1**

The Working Tree, Staging Area, and Local Repository



Il comando **git add** aggiunge una modifica all'area di staging dalla directory di lavoro. Si informa Git che **si desidera includere modifiche a un file specifico nel prossimo commit**. Tuttavia, git add ha scarso effetto sul repository: le modifiche non vengono realmente registrate finché non esegui git commit.

Il comando **git commit** salva un'istantanea delle modifiche attualmente implementate nel progetto. Gli snapshot confermati sono versioni "sicure" di un progetto che Git non modificherà mai a meno che non lo si chieda espressamente.

Working Area vs. Staging Area

- La **working area** contiene la copia in locale del proprio progetto con relative modifiche apportate, che restano qui finché non verranno aggiunte all'area di staging.
- La **staging area** contiene le modifiche che faranno parte del prossimo commit. Si possono contrassegnare sia interi file, sia singole porzioni di modifiche a un file. Se, per esempio, modifichiamo e salviamo un file, aggiungiamo il file alla staging area, modifichiamo e salviamo di nuovo il file, poi eseguiamo un commit, solo la prima delle due modifiche sarà inclusa nel commit, anche se si tratta dello stesso file.

Commit in Git

Una **commit** in Git rappresenta uno snapshot (istantanea) del repository in uno specifico momento nel tempo.

- Il commit in Git
  - memorizza le differenze tra due versioni successive del contenuto della directory,
  - salva l'istantanea (compressa) del contenuto di tutti i file che al momento del commit facevano parte del repository
- Se un file non cambia contenuto tra due commit, non viene memorizzata una nuova versione compressa, ma viene messo un riferimento al commit precedente
- Ogni commit conosce il suo "genitore" (parent commit), ossia il commit a lui immediatamente precedente
- Il commit è l'entità base dei repository Git.

schematizzando...

- **git init** => crea un repository nella directory corrente; in pratica viene creata nella directory una sotto-directory nascosta .git in cui verranno salvati di volta in volta tutti i fine necessari a Git per conoscere la cronologia delle modifiche (history) e lo stato attuale;
- **git add** => aggiunge il/i file alla mia staging Area
  - **git add nomefile** => aggiunge il singolo file
  - **git add .** => aggiunge tutti i file
  - **git *.txt** => aggiunge solo i file .txt
- **git commit -m “descrizione”** => salvare i file nel repository in Git

schematizzando...

- **git status** => vedere lo stato del repository in Git
- **git log** => vedere la storia del repository in Git, ovvero lo storico dei commit
- **git log --online** => visualizza la lista dei commit a disposizione in modo abbreviato
- **git diff =>** mostra le differenze tra l'attuale contenuto della working area e l'ultimo commit
- **git diff --staged =>** mostra le differenze tra l'attuale contenuto della staging area e l'ultimo commit
- **git diff ./path/to/file` =>** mostra le differenze solo per il file indicato
- **$ git diff 957fbc92b123030c389bf8b4b874522bdf2db72c** ce489262a1ee34340440e55a0b99ea6918e19e7a => esegue il confronto tra due commit
- **git diff feature-branch..other-feature-branch:**esegue il confronto tra due branch

Git reset [soft, mixed e hard]

A volte, quando si lavora con un repository Git, ci si rende conto che si ha bisogno di annullare le ultime modifiche, ovvero annullare l'ultimo commit.
Git fornisce diversi metodi per tornare a un commit precedente e lavorare da quel punto.
Uno degli strumenti più potenti forniti da Git per passare a uno stato precedente è il comando **Git reset**.
Ne esistono tre versioni:

- soft
- mixed (default)
- hard

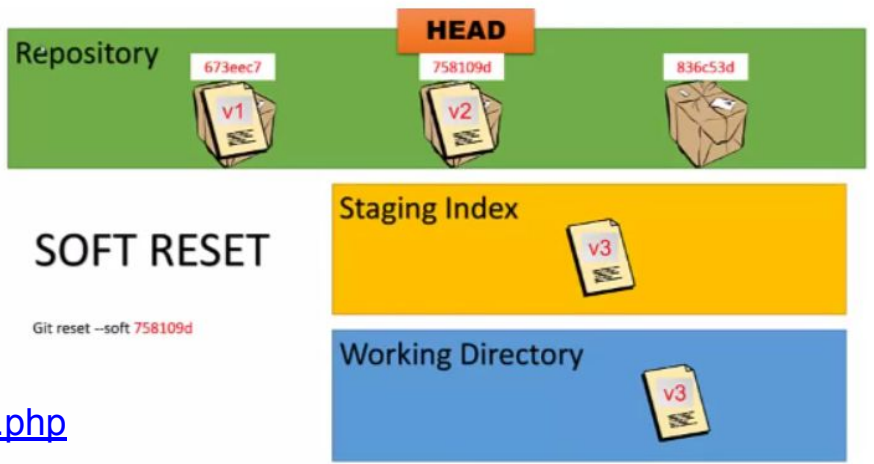
https://youtubetranscript.com/?v=V5d49WePfqo&t=591 (dal min 9.51)

- **git reset --soft idcommit**
- **git reset --soft HEAD^**

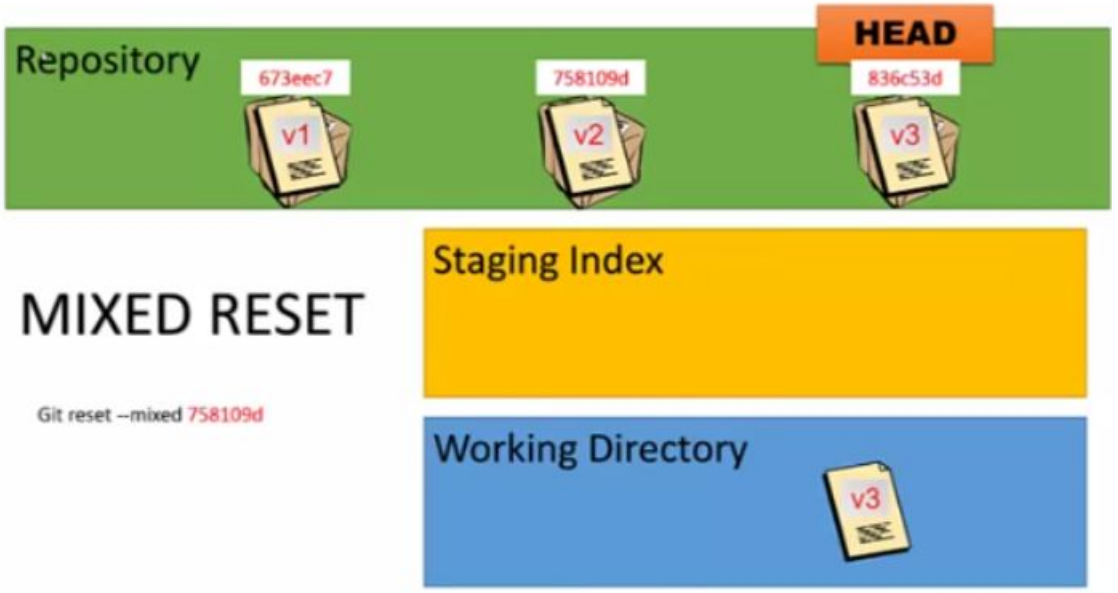
num. di passi indietro

- **git reset –soft HEAD~1**

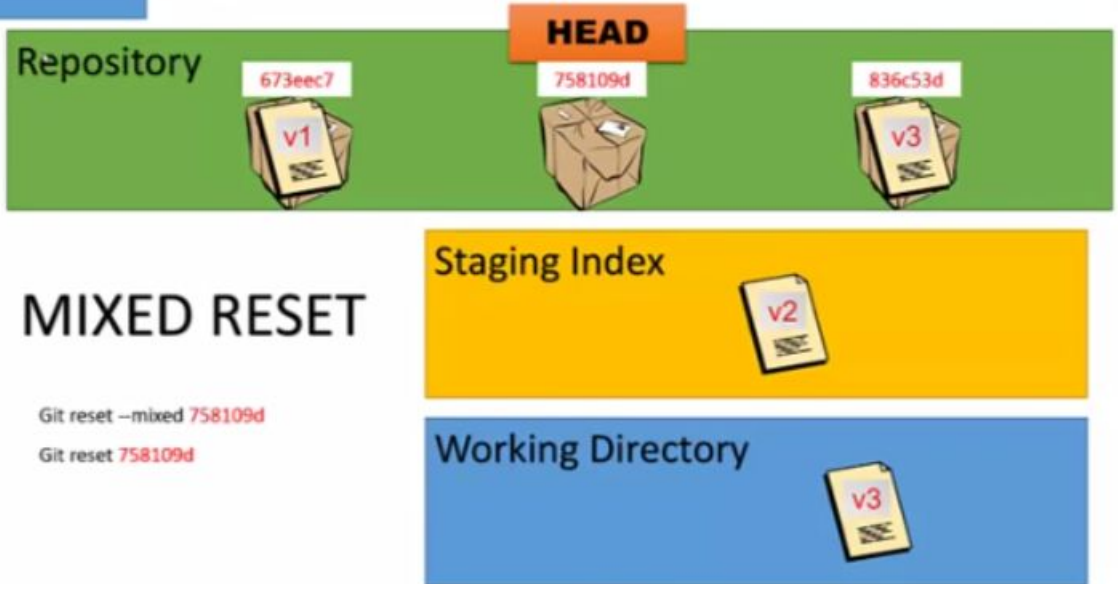
Viene **eliminata** la commit e i file vengono riportati alla **staging area** quindi appena prima del comando commit e subito dopo il comando add => si preservano le modifiche

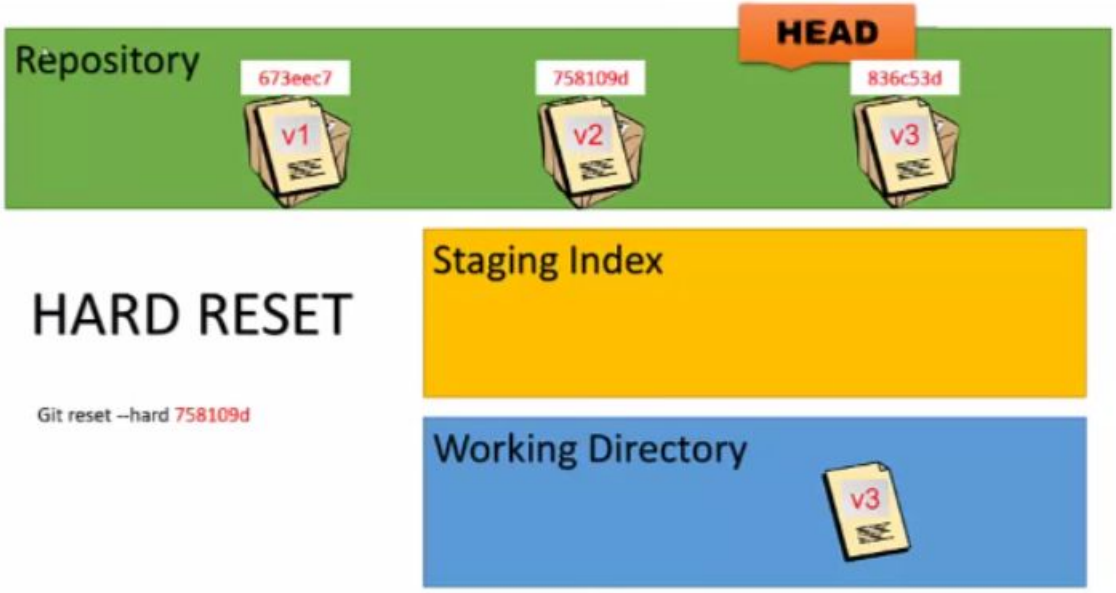


https://www.bogotobogo.com/cplusplus/Git/Git_GitHub_Soft_Reset.php
https://www.gitkraken.com/learn/git/git-reset

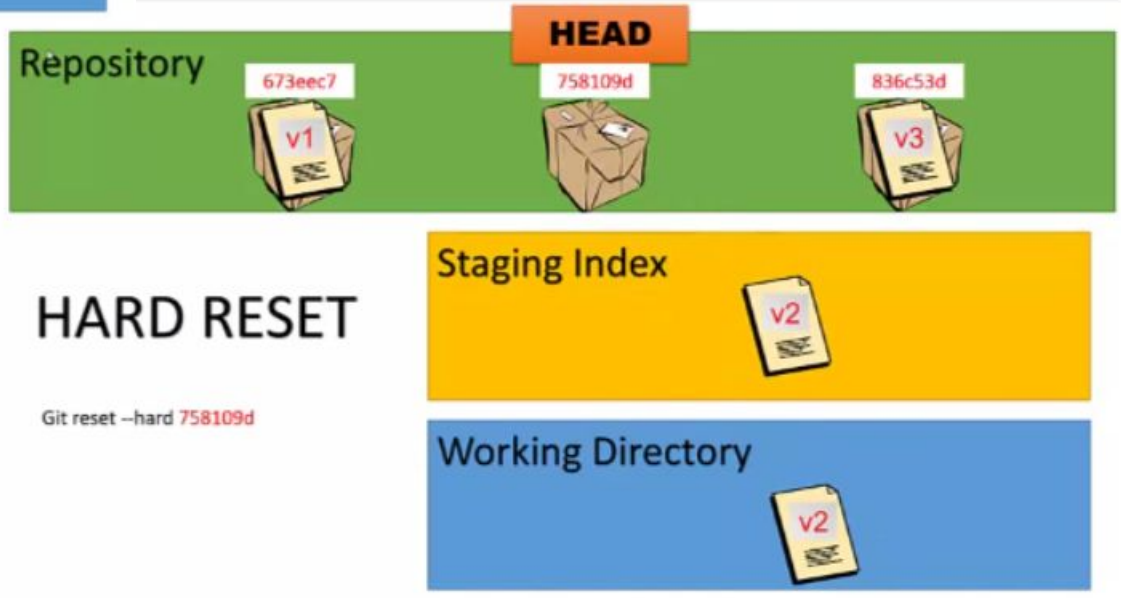


Annulla la commit e tutti i file che erano stati committati tornano nella **working directory**

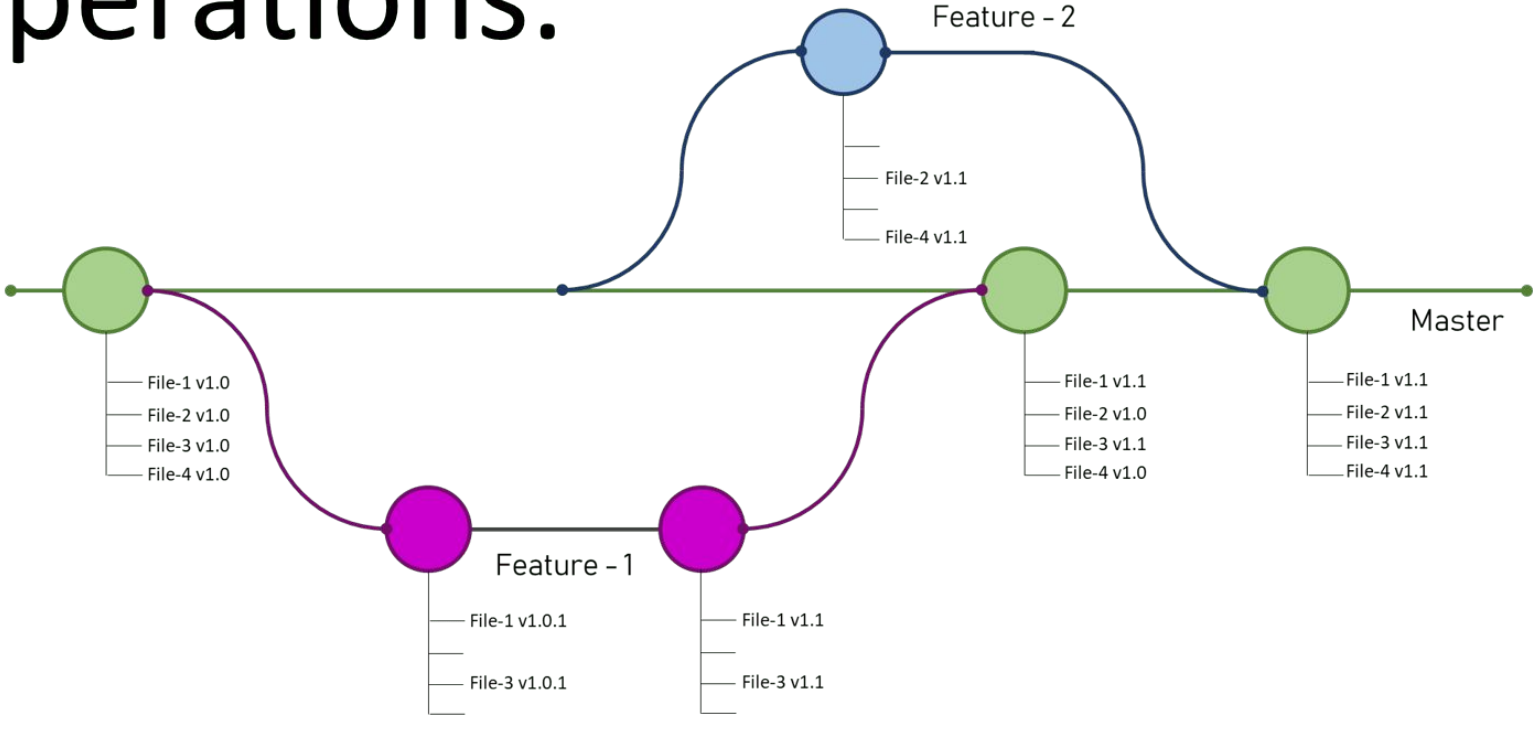




Viene resettato l'ultimo commit e si va a ripristinare lo stato della commit precedente prima che venissero fatte altre modifiche (nel nostro caso il file v3 viene eliminato)



GIT Branch and its Operations.



Git branch

Un **branch** è una **linea di sviluppo separata all'interno di un repository** e consente di fare delle modifiche senza influire sul codice principale, più stabile e adatto all'utilizzo in produzione.

In un progetto complesso, più sviluppatori possono dover lavorare contemporaneamente per implementare funzionalità diverse, per correggere bug ed effettuare test.

Per evitare di interferire con il lavoro degli altri è possibile creare un branch separato dal main branch; in questo modo tutte **le modifiche apportate a quel branch sono mantenute divise da quelle apportate agli altri**.

main o **master** sono i nomi di default dati al ramo principale dei progetti.

Git branch, checkout, merge - comandi

Comando	Descrizione
git branch *nome*	*crea un'etichetta che punta all'ultimo commit eseguito sotto quel branch, la diramazione si va a creare effettivamente quando si fanno i commit dentro lo stesso branch*
git checkout *nome*	*per spostarsi in un altro branch*
git branch	elenca tutti i branch
git checkout -b nome	*equivale a git branch nome + git checkout nome*
git merge nomeb	*effettua il merge del branch nomeb con il branch in cui ci si trova, ad esempio il main*
git merge –abort	*eliminare un merge*
git branch -d nome	elimina un branch già unito
git branch -D nome	eliminare un branch non ancora unito

Git merge

Una volta completato il lavoro e ottenuto un codice funzionale e stabile, il ramo di sviluppo può essere unito al ramo principale tramite il comando **git merge**.

Per **effettuare il merge del branch feature-X con il main**, proseguire come segue:

1- verificare di essere sul ramo principale e che HEAD punti al branch di destinazione corretto con:

   git status

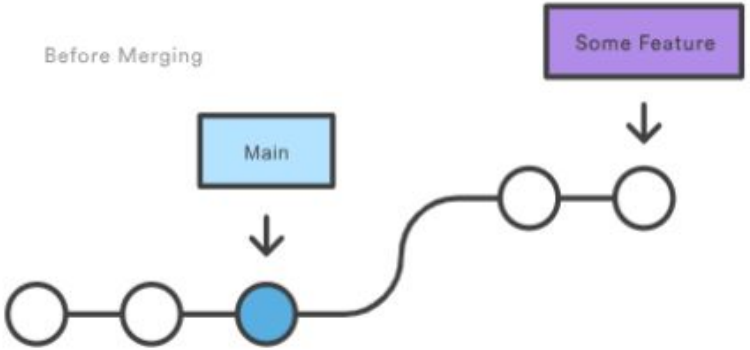
2- se necessario, posizionarsi sul ramo principale con:
   git checkout *main*

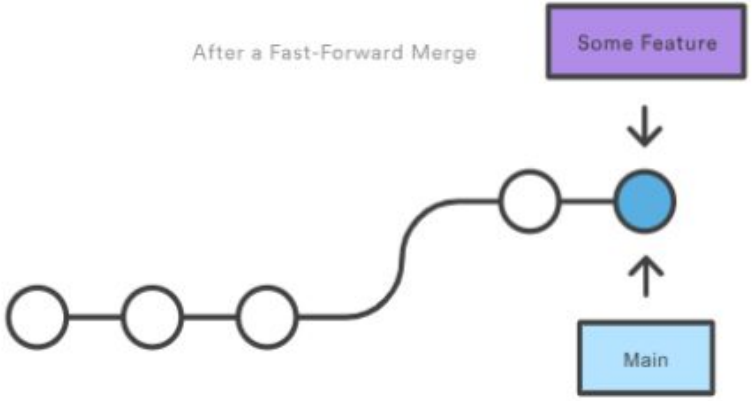
3- verificare che il branch main e il branch feature-X siano aggiornati con le ultime modifiche remote in GitHub (**opzionale**)

4- specificare il branch che si vuole unire al main, ad esempio feature-X con:
   git merge feature-X

Merge fast forward ad una via

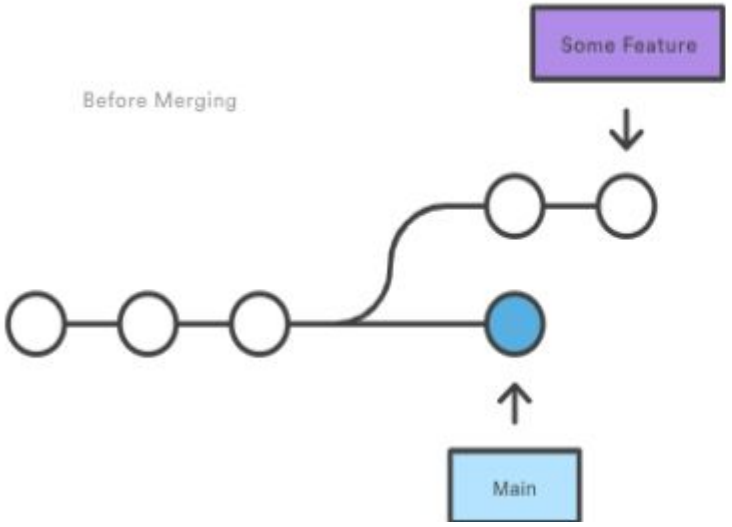
Un merge con avanzamento rapido può verificarsi **quando c'è un percorso lineare dalla punta del branch corrente al branch di destinazione**. Invece di eseguire "effettivamente" il merge dei branch, tutto ciò che Git deve fare per integrare le cronologie è spostare la punta del branch corrente fino alla punta del branch di destinazione, ovvero "effettuare un avanzamento rapido". Questa operazione unisce di fatto le cronologie, poiché tutti i commit raggiungibili dal branch di destinazione sono ora disponibili attraverso quello corrente.

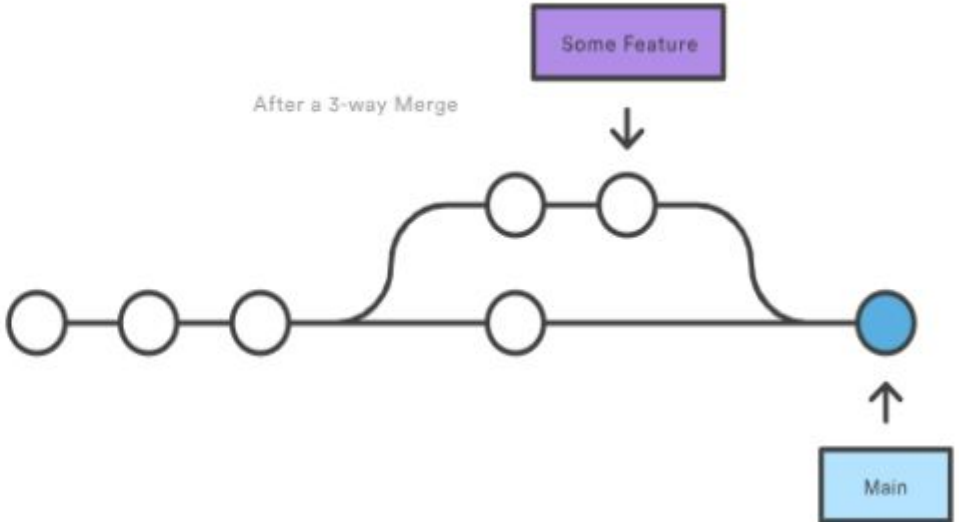




Merge a 3 vie

Tuttavia, non è possibile eseguire un merge con avanzamento rapido se i branch sono divergenti. **Quando non c'è un percorso lineare verso il branch di destinazione**, Git non ha altra scelta che unire le cronologie tramite **un merge a 3 vie**. I merge a 3 vie utilizzano un commit dedicato per legare le due cronologie. Questa nomenclatura deriva dal fatto che Git usa tre commit per generare il commit di merge: le punte dei due branch e il loro predecessore comune



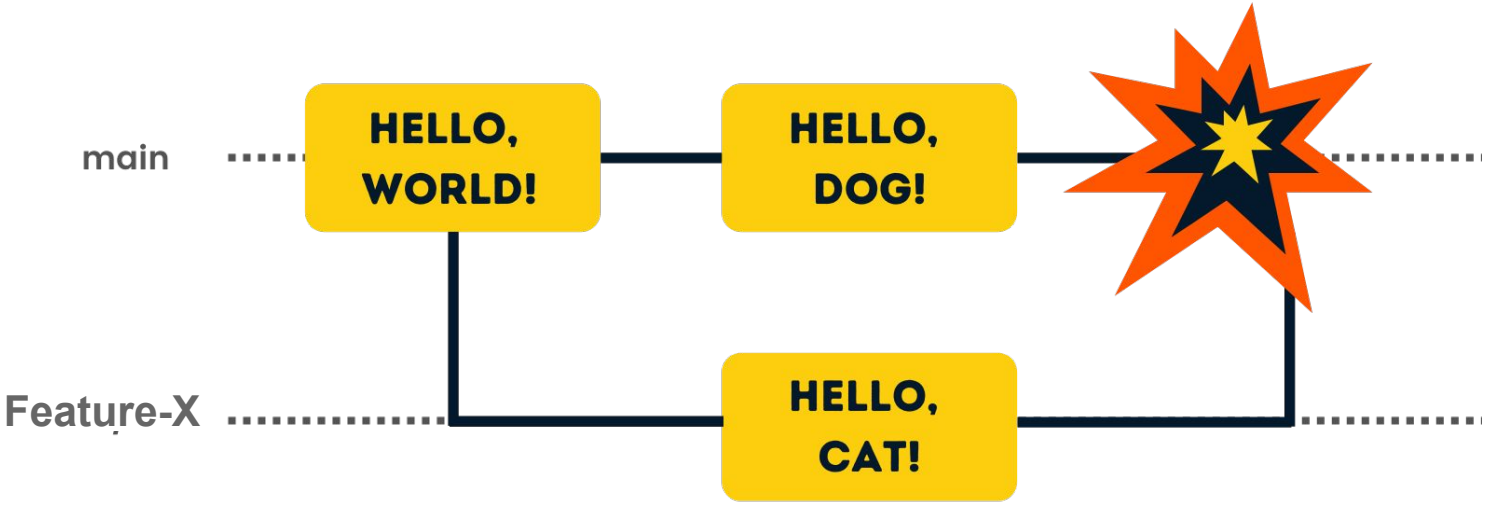


Git merge - conflitti

Con il comando **merge**, GIT cercherà di unire al main le modifiche apportate nel branch in maniera automatica, tuttavia è possibile che si verifichino dei conflitti.

Ad esempio se i due branch di cui si sta cercando di eseguire il merge hanno modificato entrambi la stessa parte dello stesso file, Git non sarà in grado di capire quale versione usare.

Git richiede all'utente di risolverli manualmente scegliendo quale modifica tenere e quale scartare.



Git merge - conflitti

Quando si verifica una situazione del genere, GIT si arresta subito prima del commit di merge, in modo da poter risolvere i **conflitti manualmente**.

Quando si verifica un conflitto di merge, eseguire il comando **git status** per visualizzare i file che devono essere risolti. Ad esempio, se entrambi i branch hanno modificato la stessa sezione di hello.py, si vedrà un output simile al seguente:

```
On branch main
Unmerged paths:
(use "git add/rm ..." as appropriate to mark resolution)
both modified: hello.py
```

Git merge - conflitti

Quando Git rileva un conflitto durante un merge, modificherà il contenuto dei file interessati con indicatori visivi che contrassegnano entrambi i lati del contenuto in conflitto.
Questi indicatori visivi sono: <<<<<<<, ======= e >>>>>>>. È utile cercare questi indicatori nel progetto durante un merge per individuare le posizioni in cui devono essere risolti i conflitti.

```
here is some content not affected by the conflict
<<<<<<< main
this is conflicted text from main
=======
this is conflicted text from feature branch
>>>>>>> feature branch;
```

Generalmente il contenuto prima dell'indicatore ======= è il branch di destinazione, mentre la parte successiva è il branch di merge.

Git merge - conflitti

Dopo aver identificato le sezioni in conflitto, si può correggere il merge a proprio piacimento.

Per terminare il merge, eseguire

1- **git add** sui file in conflitto per dire a Git che è stato risolto

2- **git commit -m** *“testo”* per generare il commit di merge.

GITHUB

Git è il sistema di controllo versione => gestisce la cronologia delle modifiche a un progetto e permette a più persone di lavorare su di esso contemporaneamente.

GitHub è una piattaforma di hosting per progetti Git => fornisce uno spazio online per archiviare, collaborare e condividere progetti Git, oltre a offrire funzionalità di gestione del progetto.

È importante notare che Git può essere utilizzato senza GitHub. Git è installato localmente sui computer degli sviluppatori e può essere utilizzato anche in un ambiente chiuso o senza connessione a Internet. GitHub, d'altra parte, è una piattaforma basata su cloud che consente di caricare i repository Git in modo che siano accessibili e collaborabili da qualsiasi posizione con accesso a Internet.

Esistono inoltre altre piattaforme simili a GitHub, come GitLab e Bitbucket.

Il comando git push

Il comando **git push** è utilizzato **per caricare i commit locali su un repository remoto**. Con questo comando aggiorniamo i file contenuti nel repository con le modifiche che abbiamo fatto in locale e facciamo in modo che gli altri membri del team con cui eventualmente stiamo collaborando possano accedervi e integrarle alla loro versione del codice.

Se non utilizziamo questo comando, le modifiche apportate verranno mantenute solo nel repository locale e non saranno accessibili da remoto.

git push
Quando si esegue il comando git push su un repository che non è di proprietà dell'utente, è necessario disporre delle autorizzazioni per accedere: se lo eseguiamo senza avere i permessi di scrittura, il comando git push restituirà un errore e non invierà le modifiche. In questo caso, potrebbe essere necessario creare un fork del repository e inviare **una pull request** per richiedere (o suggerire!) l'integrazione delle modifiche

Il comando git pull

Il comando **git pull** esegue l'operazione inversa a quella di git push: è utilizzato per recuperare le modifiche **dal repository remoto e integrarle con il repository locale** in modo da avere l'ultima versione del progetto.

git pull
Se non utilizziamo il comando quando lavoriamo a un progetto in collaborazione con altri, il repository locale potrebbe diventare obsoleto rispetto a quello remoto e potrebbero verificarsi conflitti tra le modifiche apportate.

Pull request

Una pull request è una richiesta che un collaboratore invia al proprietario del repository per integrare le modifiche apportate in un branch in una branch principale.

Fork del Repository: dal repository del progetto sul quale vuoi lavorare, clic sul pulsante **"Fork"** nella parte superiore destra della pagina. Questo creerà una copia del repository sul tuo account GitHub.

Clonare il Repository: dall'account personale di GitHub, trovare la copia del repository (il proprio fork) e clonarlo sul proprio computer usando il comando **git clone.**

Creare un Branch e apportare le modifiche necessarie al codice

Commit e Push

Creare la Pull Request: andare nel proprio fork su GitHub.
Individuare il branch e cliccare sul pulsante "Confronta e pull request".
Assicurare che le modifiche siano corrette, aggiungere una descrizione e fai clic su "Create pull request".

Revisione e Merge:
I collaboratori del progetto possono esaminare le modifiche, fare commenti e discutere eventuali cambiamenti necessari.
Una volta che la pull request è stata approvata, **il proprietario del repository** può eseguire il merge delle tue modifiche nel branch principale.

Il comando git fetch

Abbiamo visto che il comando git pull sincronizza il repository remoto con quello che abbiamo in locale. Se vogliamo solo scaricare e visualizzare le modifiche senza applicarle automaticamente, usiamo il comando git fetch (in italiano "raggiungere" o "andare a prendere"):

git fetch

Se accettiamo le modifiche e vogliamo applicarle alla copia locale possiamo usare semplicemente il comando git pull.

Il comando git clone

Il comando **git clone** consente di scaricare l'intero repository dal repository remoto e creare una copia locale sul proprio computer. Questa copia sarà comprensiva di tutti i file, la cronologia dei commit e le informazioni di configurazione in un determinato momento dello sviluppo del progetto. È possibile usare il comando git clone sia sui propri repository che su quelli pubblici di altri developer.

Quando si utilizza il comando bisogna **specificare l'URL del repository remoto** che si vuole clonare. Ad esempio, creiamo una copia locale del repository "jerry" da quello remoto su GitHub:

git clone https://github.com/username/jerry.git
Su GitHub possiamo ottenere il comando sopra descritto automaticamente, cliccando sull'apposito tasto verde.

Quando si clona un repository **viene creato un collegamento tra il repository locale e quello remoto, identificato dal nome origin (in italiano “origine”),** che è il termine usato per riferirsi al repository remoto da cui è stato clonato il repository locale. Sarà possibile esplorare il codice e i file del progetto, così come anche eseguire il codice, come se ci si trovasse nel computer degli sviluppatori.

ESERCIZIO

Come usare Git e GitHub in un team come un professionista – Con la partecipazione di Harry e Hermione