

# Un approccio pratico a **Git**

by Paolini Arianna

Questa guida si propone di fornire i concetti base per capire il funzionamento del software per il controllo di versione **Git** e facilitarne l'utilizzo durante la realizzazione di progetti in singolo o in gruppo, con esempi pratici e riferimenti alla piattaforma **GitHub**.

## Sommario

Cos'è <b>Git</b> ?	2
Git vs <b>GitHub</b>	3
Struttura dati di Git	3
Oggetti Git e DAG	3
Riferimenti	5
Operazioni	6
Commit	6
Branching	6
Merging	7
Rebasing	8
Comandi <b>git</b>	9
Creare un repository	9
Fare un commit	10
Recuperare le versioni precedenti del progetto	12
Creare e unire rami di sviluppo	13
Risolvere merge conflicts	14
Lavorare in gruppo	15
Connettersi a un repository remoto	15
Fare pull e push	17
Gestire un progetto con Git e <b>GitHub</b>	19
Caricare un repository locale su <b>GitHub</b>	19
Scaricare un repository <b>GitHub</b> in locale	20
Collaborare	22
<b>GitHub</b> flow	22
Pull requests	23
Forks	25
Link utili	27

## Cos'è Git?

Quante volte ti è capitato di cambiare il codice di una classe Java con l'intento di migliorarlo per poi finire solo col creare problemi alle funzionalità pre-esistenti? Con Git avresti potuto facilmente far finta di nulla e tornare velocemente alla versione precedente del progetto, senza dover cercare e correggere a mano tutte le linee di codice modificate.

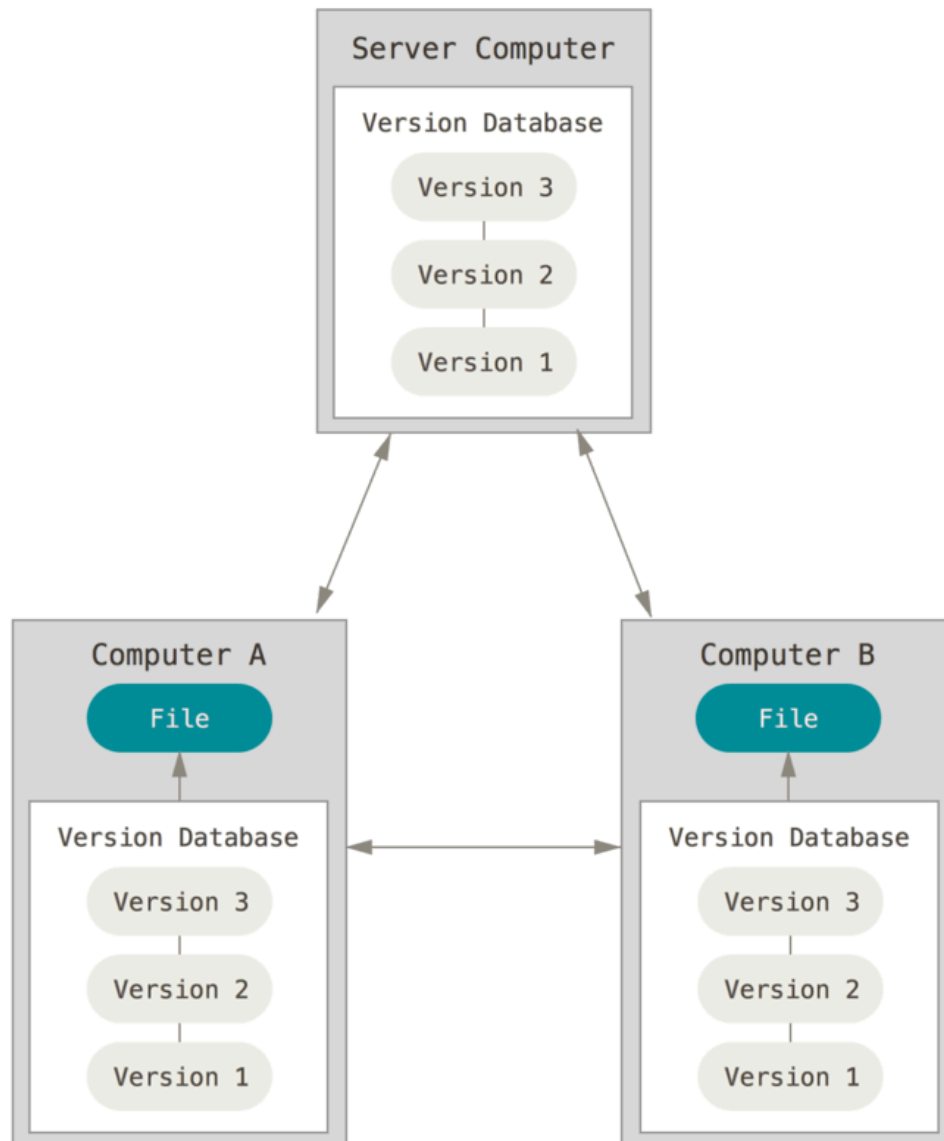


Figura 1:

Git è un VCS distribuito, ogni client ha l'intero backup di tutti i dati

Git infatti è un **VCS** (*Version Control System*), ovvero, in parole povere, un software in grado di tenere traccia delle modifiche eseguite su un insieme di file. Lavorando ad un progetto è quindi possibile mantenere nel tempo la storia delle sue versioni passate e, in caso di necessità, recuperarle.

Inoltre Git facilita la **collaborazione** di più sviluppatori allo stesso progetto, offrendo meccanismi per sincronizzare le modifiche su file conservati in un server remoto condiviso. Poiché Git è un **sistema distribuito** (Figura 1), ogni collaboratore avrà una copia locale di tutte le versioni del progetto, riducendo il rischio di perdite di dati.

## Git vs GitHub

Contrariamente a quanto spesso si pensa, Git e GitHub non sono la stessa cosa. Come detto sopra, Git è un VCS, mentre GitHub è una piattaforma online che offre un'interfaccia grafica alle funzionalità di Git (non è l'unica ma sicuramente la più utilizzata).



GitHub è di fatto un sito su cui creare account, creare e conservare progetti, gestire le modifiche su di essi e le collaborazioni con altri utenti; spesso funge da «magazzino» in cui trovare software da scaricare.

Lavorando in gruppo, il progetto condiviso sarà ospitato da GitHub, su cui ognuno potrà sincronizzare, tramite specifiche operazioni Git, le modifiche realizzate prima sulla propria copia locale.

## Struttura dati di Git

Per poter capire veramente Git (e usarlo più facilmente) è necessario conoscere il modello a grafo utilizzato per mantenere i dati (non ti spaventare, è abbastanza intuitivo).

### Oggetti Git e DAG

Git memorizza la storia delle versioni di un insieme di file e cartelle come una sequenza di «snapshots» (o «istantanee») della cartella principale (alla radice del progetto) al momento corrente. I file vengono denominati *blob* mentre le cartelle sono chiamate *tree*. Esse mappano i nomi di file e cartelle da loro contenute ai corrispondenti blobs e trees (Figura 2).

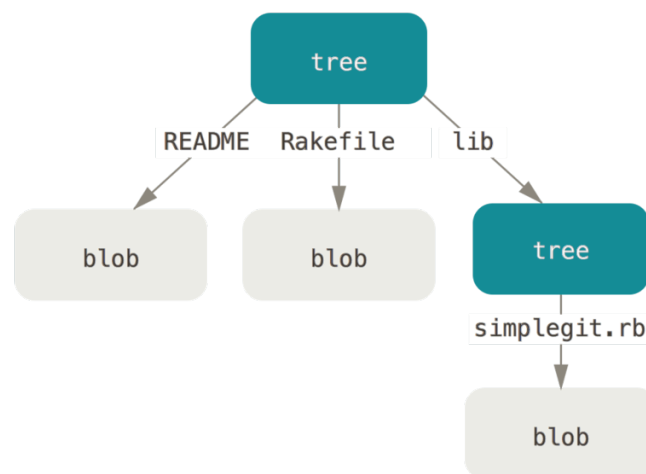


Figura 2: Illustrazione concettuale del modello dei dati di Git: la cartella top-level contiene due file («README» e «Rakefile») e una sotto-cartella «lib», che a sua volta contiene un file «simplegit.rb»

I vari snapshots del progetto sono rappresentati da oggetti chiamati *commits* e vengono organizzati in un *DAG* (grafo diretto e aciclico, Figura 3) che semplicemente associa con una freccia ogni commit ai suoi «genitori», cioè i

commits immediatamente precedenti (il commit può avere più di un genitore perché, ad esempio, come sarà spiegato più avanti, è possibile fondere due rami paralleli di sviluppo tramite un'operazione di *merge*).

Commits, blobs e trees formano gli oggetti (*objects*) principali utilizzati da Git nel suo modello dati.

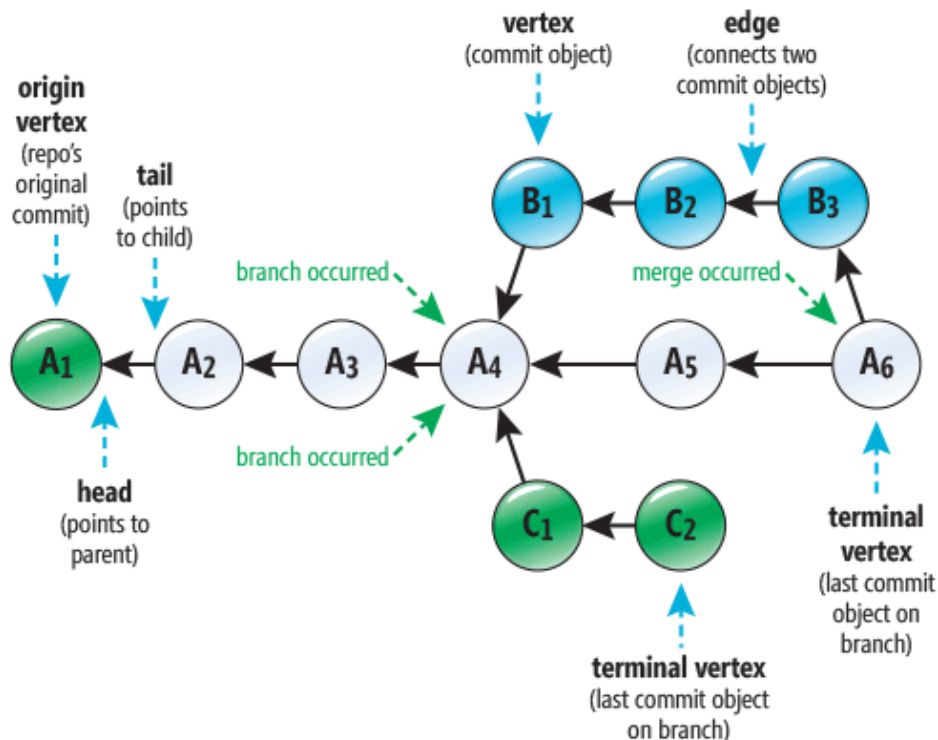


Figura 3:

Illustrazione del DAG di Git: i commits sono i nodi (o vertici) del grafo, mentre gli archi legano i commit ai propri genitori. Un commit con più figli è legato a un'azione di *branching*, mentre un commit con più genitori corrisponde ad un'azione di *merging*.

I commit sono immutabili: non si possono modificare. Quando si cambia qualcosa nei file di un progetto si possono creare nuovi commit da aggiungere al DAG, ma senza interferire con quelli pre-esistenti. Oltre al riferimento ad una specifica versione del progetto (sotto forma di tree), un commit contiene meta-dati come il suo autore e un messaggio descrittivo. Se volessimo rappresentarlo con una classe Java, scriveremmo qualcosa tipo:

```
class Commit {  
    List<Commit> genitori;  
    String autore;  
    String messaggio;  
    Tree snapshot;  
    ...  
}
```

Il *tag* è un ulteriore tipo di oggetto Git. Esso contiene il riferimento ad un commit e viene utilizzato per annotarlo con un nome ed eventualmente un messaggio (ad esempio si usa spesso taggare i commit con nomi di versione come «v1.0», «v2.0», ecc.).

## Riferimenti

Per utilizzare un oggetto Git non si richiede ogni volta l'intera copia del suo contenuto in memoria ma si sfrutta il **riferimento** all'unica copia presente nello store generale degli oggetti di Git (Figura 4), identificata dal suo **hash SHA-1** (una stringa esadecimale di dimensione fissa ottenuta dall'applicazione di una funzione, detta *funzione di hash*, sull'oggetto stesso).

Poiché per noi umani è difficile leggere e ricordare lunghe stringhe di caratteri esadecimali, Git fornisce la possibilità di usare stringhe con nomi semplici per riferirsi a particolari oggetti nello store (ad esempio di solito si usa il nome «*main*» per riferirsi all'ultimo commit del ramo principale di sviluppo del proprio progetto).

In pratica si crea una mappa da stringhe con contenuto comprensibile all'uomo a stringhe esadecimali che rappresentano gli identificativi degli oggetti (Figura 5).

Questo tipo di riferimento è mutabile: si può riassegnare un nome semplice ad un nuovo valore esadecimale.

**Object Store**

Object ID	Object Data
31302b38...	commit...
6a475b71...	commit...
7ce4f05b...	blob...
84fbf166...	commit...
9771dc66...	tree...
a19e856f...	tree...
a4a2aa60...	tree...
b520aa9c...	blob...
e3464c2c...	commit...
e4a4b315...	commit...
ffaf52ec...	tag...

Figura 4:

Rappresentazione dell'object store di Git: una tabella di oggetti dove le chiavi sono i rispettivi hash

**Reference Store**

Reference Name	Object ID
refs/heads/main	84fbf166...
refs/tags/v2.37.0	ffaf52ec...
refs/remotes/origin/main	6a475b71...
refs/remotes/origin/next	e3464c2c...

**Object Store**

Object ID	Object Data
31302b38...	commit...
6a475b71...	commit...
7ce4f05b...	blob...
84fbf166...	commit...
9771dc66...	tree...
a19e856f...	tree...
a4a2aa60...	tree...
b520aa9c...	blob...
e3464c2c...	commit...
e4a4b315...	commit...
ffaf52ec...	tag...

Figura 5:

Rappresentazione della tabella di riferimenti che mappa stringhe con nomi semplici a identificativi di oggetti Git

Esiste un riferimento speciale per indicare il commit corrente (quello che diventerebbe il genitore di un eventuale nuovo commit), indicato come «**HEAD**»: esso punta alla posizione attuale all'interno del grafo dei commits.

## Operazioni

La struttura a grafo di Git appena discussa può essere modificata con le **operazioni concettuali** presentate di seguito, di cui si forniranno i comandi per l'esecuzione nella sezione successiva.

### Commit

L'operazione più semplice è il **commit**: si aggiunge uno snapshot dei contenuti attuali del progetto al grafo con i commits precedenti, creando un nuovo nodo che avrà come genitore il commit puntato dal riferimento **HEAD** al momento della creazione. Dopo l'aggiunta del nuovo commit, **HEAD** si sposterà automaticamente su di esso:

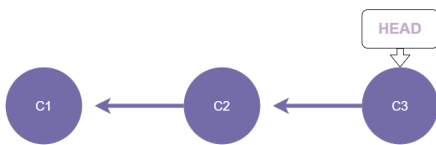


Figura 6: Ipotetico grafo *G* di partenza, contenente i commit C1, C2, C3

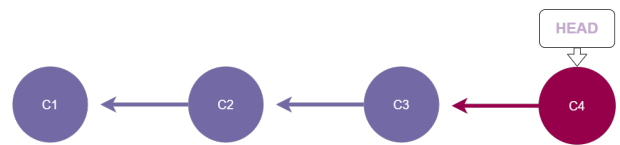


Figura 7: Grafo *G* dopo l'aggiunta del commit C4 tramite l'operazione di *commit*

### Branching

Git permette di creare diversi **rami di sviluppo** tramite un'operazione di **branching**, per facilitare ad esempio il lavoro su progetti di gruppo o l'implementazione di funzionalità in parallelo. Nella struttura a grafo, ciò corrisponde ad osservare un nodo con più figli, ciascuno su un ramo diverso.

Ogni ramo è identificato da un **puntatore** con nome (ad esempio «*main*»): esso specifica l'ultimo commit del ramo, cioè il punto in cui aggiungere eventuali nuovi commits per il ramo stesso.

Per poter lavorare su un ramo in particolare bisogna assicurarsi che «**HEAD**» punti su di esso, eseguendo se necessario un'operazione di **switch** dal ramo correntemente puntato a quello desiderato.

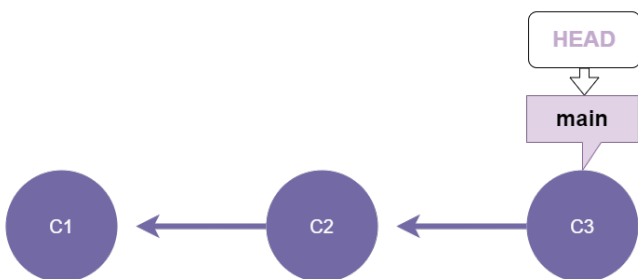


Figura 8:

Ipotetico grafo *G1* di partenza, contenente i commit C1, C2, C3 sul ramo *main*

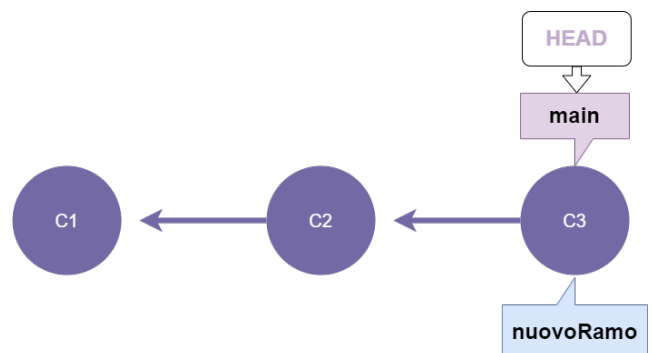


Figura 9:

Grafo *G2* ottenuto dal grafo *G1* dopo la creazione del ramo *nuovoRamo* tramite operazione di *branching*

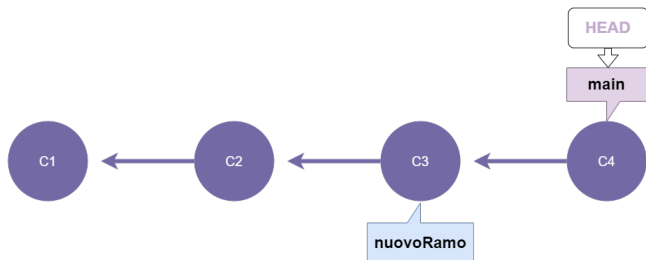


Figura 10:

Grafo *G3* ottenuto dal grafo *G2* dopo l'aggiunta del commit *C4* tramite operazione di *commit* (l'aggiunta avviene sul ramo *main* perché puntato da *HEAD* in *G2*)

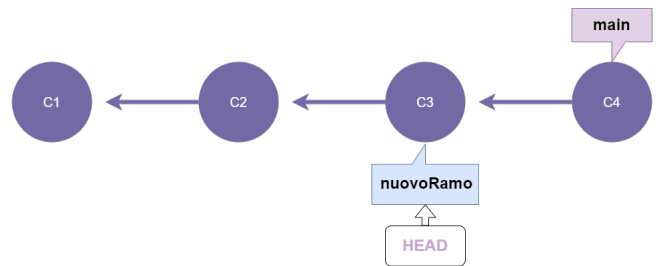


Figura 11:

Grafo *G4* ottenuto dal grafo *G3* dopo lo switch da *main* a *nuovoRamo* (si sposta il puntatore *HEAD*)

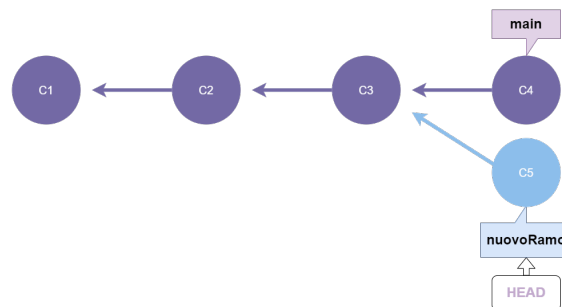


Figura 12:

Grafo *G5* ottenuto dal grafo *G4* dopo l'aggiunta del commit *C5* tramite operazione di *commit* (l'aggiunta avviene sul ramo *nuovoRamo* perché puntato da *HEAD* in *G4*)

## Merging

L'operazione di *merging* può essere considerata quasi «inversa» a quella di *branching*, in quanto permette di fondere insieme rami diversi, unendo i cambiamenti da essi apportati ai file del progetto. Talvolta ciò dà origine a un nodo con più genitori nel grafo dei commits. Il *merge* può essere utilizzato, ad esempio, per ritornare ad un'unica linea di sviluppo dopo aver testato e messo insieme le funzionalità implementate in precedenza su rami paralleli.

Quando si vuole fondere un ramo *B* in un ramo *A* il cui ultimo commit si trova sul percorso a ritroso dall'ultimo commit di *B* alla radice del grafo (cioè se il commit di *A* è un antenato del commit di *B*), Git esegue il *merge* semplicemente spostando il puntatore del ramo *A* sull'ultimo commit di *B* (questo procedimento è denominato «*fast-forward*»):

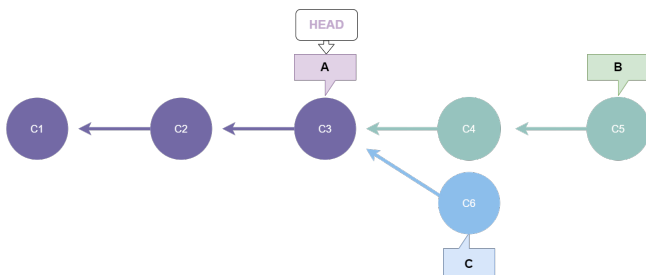


Figura 13:

Ipotetico grafo di partenza *G*, con rami *A*, *B* e *C* (il ramo *A* è correntemente selezionato)

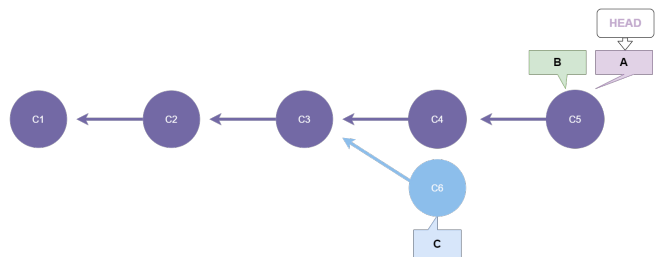


Figura 14:

Grafo *G* dopo il *merging* di *B* in *A* tramite *fast-forward*: il puntatore del ramo *A* viene spostato sul commit puntato dal ramo *B*

Se invece si desidera fondere due rami i cui ultimi commits sono su linee di sviluppo divergenti (cioè se nessuno dei due commit è antenato dell'altro), Git esegue una fusione dei due commits e del loro antenato comune più recente, generando un nuovo commit che avrà come genitori i commits puntati dai rami fusi.

Se però si tenta di fondere due rami divergenti in cui è stata modificata la stessa parte di uno stesso file del progetto in modi diversi, Git non potrà eseguire la fusione automaticamente e chiederà all'utente di risolvere manualmente i conflitti generati (*merge conflicts*), scegliendo quali modifiche mantenere e quali eliminare.

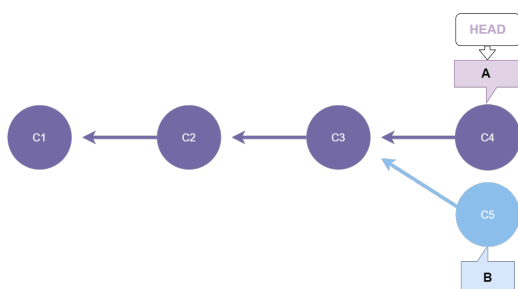


Figura 15:

Ipotetico grafo di partenza G, con rami A e B (il ramo A è correntemente selezionato)

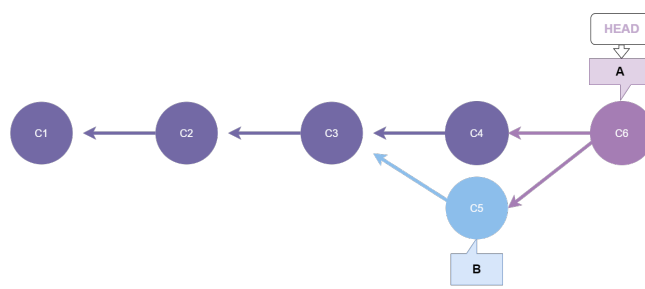


Figura 16:

Grafo G dopo il *merging* di B in A e la generazione del *merge commit* C6 tramite la fusione di C3, C4 e C5

## Rebasing

Un'operazione alternativa al *merging* è il *rebasing*, che consiste nel replicare i cambiamenti apportati al progetto da un ramo di sviluppo A a partire dall'ultimo commit di un ramo B, «spostando» il ramo A in modo tale da accodarsi a B:

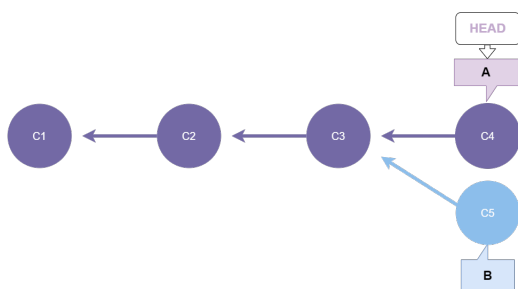


Figura 17:

Ipotetico grafo di partenza G, con rami A e B (il ramo A è correntemente selezionato)

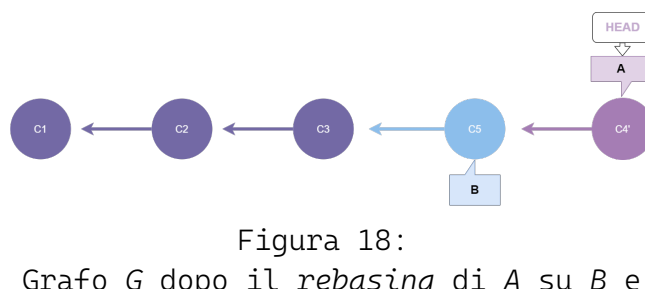


Figura 18:

Grafo G dopo il *rebasing* di A su B e la generazione del *commit* C4' tramite la fusione di C3, C4 e C5

Il commit finale risultante dal *rebasing* include una fusione del contenuto del primo antenato comune tra i rami A e B e dei loro rispettivi ultimi commits, quindi il risultato è lo stesso che si otterrebbe con un *merge*. Ciò che cambia è la forma del grafo, cioè della **storia delle versioni** del progetto: nel *merge* appare un commit di fusione come un nodo con più genitori, nel *rebase* la storia è lineare e ordinata, facendo apparire le modifiche come se fossero avvenute in modo sequenziale, nonostante siano state in realtà introdotte in parallelo.



## Comandi **git**

Di seguito vengono illustrati alcuni comandi per usare Git da terminale ed eseguire, tra le varie azioni, anche le operazioni presentate nella sezione precedente.

Per iniziare, installare Git ([pagina download](#)) e verificare da linea di comando (su un terminale di sistema) che l'installazione sia andata a buon fine, usando il comando **git --version** per visualizzare la versione di Git presente sul proprio computer.

```
PS C:\> git --version
git version 2.40.0.windows.1
```

Per ottenere informazioni sui comandi Git esistenti è sempre possibile usare **git --help**, mentre **git help nome\_comando** fornisce la documentazione per il comando **nome\_comando**.

## Creare un repository

Un **repository** Git è un deposito in memoria per gli oggetti e i riferimenti Git (spiegati nel capitolo precedente) del proprio progetto.

Si può creare un repository a partire da una **cartella sul filesystem locale** e abilitare così il controllo di versione Git su di essa: posizionarsi nella cartella scelta ed eseguire il comando **git init**.

```
PS C:\Users\arian\ProgettoGit> git init
Initialized empty Git repository in C:/Users/arian/ProgettoGit/.git/
```

Ciò genera una nuova sotto-cartella nascosta, denominata «**.git**», contenente i file necessari per mantenere il nuovo repository.

In alternativa è possibile ottenere un repository locale clonando un **repository pre-esistente**, ad esempio da GitHub, ma di questo si parla in una apposita sezione successiva.

## Fare un commit

Una volta creato un repository, si può scegliere di memorizzare nella storia delle versioni del progetto i cambiamenti apportati ad uno o più file nella cartella considerata. Git, infatti, mette a disposizione una [staging area](#) (Figura 19), cioè una zona di memoria temporanea per selezionare i file che si desidera includere nel prossimo commit.

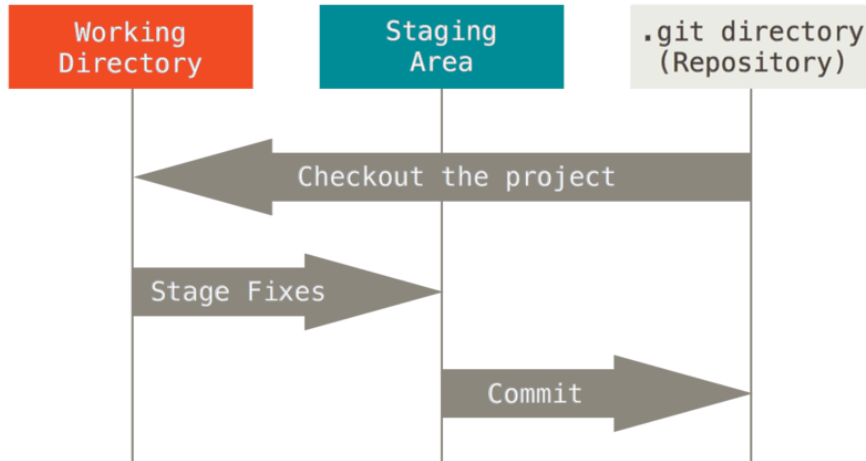


Figura 19:

Rappresentazione delle tre sezioni principali di un progetto Git: la cartella di lavoro del progetto (*working directory*), la *staging area*, dove è possibile selezionare alcuni file dalla *working directory* («*stage fixes*») e la cartella *.git* (il *repository*), dove vengono salvati i file presenti nella *staging area* al momento del *commit*

In generale, un file di un progetto con controllo di versione viene definito «**untracked**» («non tracciato») se non fa parte dell'ultimo snapshot del progetto e non è incluso nella staging area, «**tracked**» altrimenti.

Un file *tracked* può trovarsi negli stati:

**modified** il file è stato modificato ma i cambiamenti non sono ancora stati salvati nel repository Git;

**staged** il file è stato modificato e incluso nella *staging area* (ovvero è tra i file pronti per essere salvati nel repository con il prossimo commit);

**unmodified / committed** il file è al sicuro, salvato nel repository.

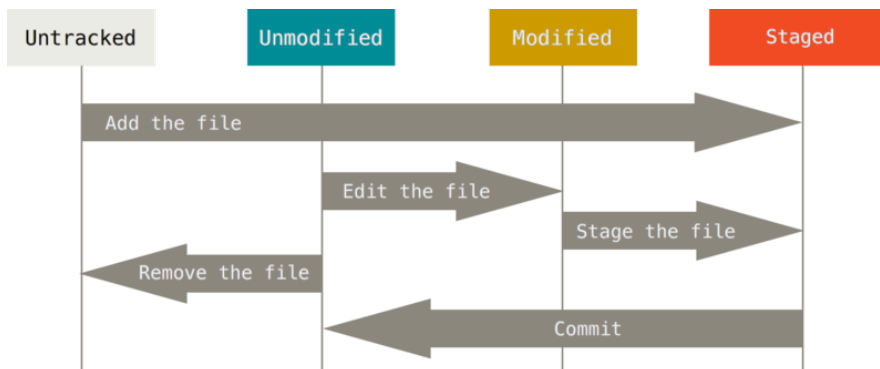


Figura 20:

## Rappresentazione degli stati di un file in un repository Git

Il comando `git status` permette di visualizzare lo stato della cartella di lavoro (cioè gli stati dei file in essa contenuti) e il contenuto della staging area. Di seguito un esempio con un repository ancora senza commit e una cartella di lavoro contenente tre nuovi file di testo («testoA.txt», «testoB.txt», «testoC.txt») non inclusi nella staging area (quindi in stato *untracked*):

```
PS C:\Users\arian\ProgettoGit> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    testoA.txt
    testoB.txt
    testoC.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Con `git add "nome_file"` si può aggiungere il file con nome *nome\_file* all'area di staging.

```
PS C:\Users\arian\ProgettoGit> git add "testoA.txt"
PS C:\Users\arian\ProgettoGit> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   testoA.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    testoB.txt
    testoC.txt
```

Scrivendo `git add .`, invece, si selezionano automaticamente tutti i file della cartella di lavoro per l'inclusione nella staging area.

```
PS C:\Users\arian\ProgettoGit> git add .
PS C:\Users\arian\ProgettoGit> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   testoA.txt
    new file:   testoB.txt
    new file:   testoC.txt
```

Tramite il comando `git commit -m "messaggio"` si genera un commit contenente le versioni correnti dei file nella staging area. Esso avrà descrizione testuale *messaggio* e un hash esadecimale identificativo. Il commit corrisponde a un nodo nel grafo della struttura dati Git, posizionato sul ramo selezionato (nell'esempio seguente il ramo «*master*», l'unico presente):

```
PS C:\Users\arian\ProgettoGit> git commit -m "primo commit"
[master (root-commit) f29f717] primo commit
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 testoA.txt
 create mode 100644 testoB.txt
 create mode 100644 testoC.txt
PS C:\Users\arian\ProgettoGit> git status
On branch master
nothing to commit, working tree clean
```

Per visualizzare la storia delle versioni del progetto come un grafo di commits si può usare `git log --graph`. Per l'esempio è stato usato un repository con 3 commits sullo stesso ramo (*master*), quindi vengono visualizzati in un'unica sequenza verticale, dal più recente (puntato da *HEAD*) al più vecchio:

```
PS C:\Users\arian\ProgettoGit> git log --graph
* commit 78c230c356d8cb9977552ab3daacbc1f0bdcfb1b (HEAD -> master)
  Author: Arianna Paolini <[redacted]@gmail.com>
  Date:   Sun Sep 15 12:37:42 2024 +0200

    terzo commit

* commit 899c25564efbb2df79c34dc66012bf309c9eae40
  Author: Arianna Paolini <[redacted]@gmail.com>
  Date:   Sun Sep 15 12:37:12 2024 +0200

    secondo commit

* commit f29f717b27edd3b7672d759290546dc2c9b2a882
  Author: Arianna Paolini <[redacted]@gmail.com>
  Date:   Sun Sep 15 12:02:03 2024 +0200

    primo commit
```

## Recuperare le versioni precedenti del progetto

Supponiamo di aver eseguito diverse operazioni di commit sul repository Git del nostro progetto e di renderci conto ad un certo punto di voler «annullare» le ultime modifiche e tornare ad una versione passata. Come si fa?

Il comando `git checkout id_commit` permette di selezionare un commit dalla storia delle versioni del progetto tramite il suo codice esadecimale identificativo `id_commit` (che può essere visualizzato con `git log`), muovendo il riferimento *HEAD* su di esso. Nell'esempio viene scelto il secondo di una serie di tre commits sullo stesso ramo:

```
PS C:\Users\arian\ProgettoGit> git checkout 899c25564efbb2df79c34dc66012bf309c9eae40
Note: switching to '899c25564efbb2df79c34dc66012bf309c9eae40'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 899c255 secondo commit
PS C:\Users\arian\ProgettoGit> git status
HEAD detached at 899c255
nothing to commit, working tree clean
```

Se il commit selezionato non è l'ultimo commit di un ramo di sviluppo, l'operazione appena descritta genera una situazione segnalata da Git come *detached HEAD state* (Figura 21), in cui eventuali nuovi commits non verrebbero associati a nessun ramo e sarebbero perciò difficili da trovare in futuro (si dovrebbero ricordare a memoria i loro codici hash).

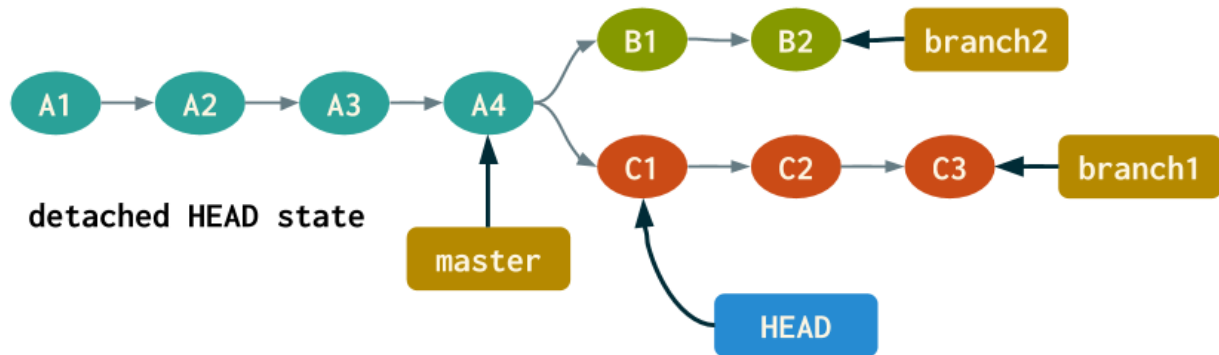


Figura 21:

Esempio di detached HEAD state: è stato selezionato un commit (C1) che non è posizionato alla fine di alcun ramo di sviluppo

Per uscire dallo stato di *detached head*:

- se si desidera continuare a lavorare sul commit corrente, si può creare un nuovo ramo, in modo che i commits seguenti siano associati ad esso;
- se si vuole abbandonare il commit corrente senza modifiche, si può tornare ad un ramo pre-esistente.

La creazione e gestione dei rami di sviluppo è spiegata nella sezione seguente.

## Creare e unire rami di sviluppo

Il comando `git branch` permette di visualizzare i rami presenti nel proprio progetto Git.

```
PS C:\Users\arian\ProgettoGit> git branch
* master
```

Uno dei modi per creare un nuovo ramo è utilizzare `git branch nome_ramo`. Per poter lavorare su di esso (cioè far sì che i commits seguenti ne facciano parte) sarà necessario selezionarlo con `git checkout nome_ramo`, facendo sì che il riferimento `HEAD` venga spostato su di esso.

```
PS C:\Users\arian\ProgettoGit> git branch ramoA
PS C:\Users\arian\ProgettoGit> git branch
* master
  ramoA
PS C:\Users\arian\ProgettoGit> git checkout ramoA
Switched to branch 'ramoA'
PS C:\Users\arian\ProgettoGit> git branch
  master
* ramoA
```

Un'alternativa è `git checkout -b nome_ramo`, che crea un nuovo ramo e lo seleziona in un unico passaggio.

```
PS C:\Users\arian\ProgettoGit> git checkout -b ramoB
Switched to a new branch 'ramoB'
PS C:\Users\arian\ProgettoGit> git branch
  master
  ramoA
* ramoB
```

Nel momento in cui si decide di unire due rami di sviluppo con un'operazione di *merging*, è possibile selezionare un ramo «target» tramite `git checkout` e usare il comando `git merge nome_ramo`, dove `nome_ramo` identifica un ramo «source» da fondere

in *target*: sarà il riferimento del ramo *target* ad essere aggiornato dopo il *merge* per includere i dati provenienti da entrambi i rami.

Nell'esempio, il ramo *ramoA* viene scelto come *target* in cui fondere il ramo *ramoB*.

```
PS C:\Users\arian\ProgettoGit> git checkout ramoA
Switched to branch 'ramoA'
PS C:\Users\arian\ProgettoGit> git merge ramoB
Merge made by the 'ort' strategy.
 testoA.txt | 2 +-
 testoB.txt | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)
```

Se invece si opta per l'unione dei rami tramite *rebasing*, ci si può posizionare sul ramo che si desidera «spostare» (nell'esempio, il ramo *ramoB*) tramite *git checkout*, e digitare *git rebase* nome\_ramo, dove nome\_ramo è il ramo a cui «appendere» il risultato del *rebase* (nell'esempio, il ramo *ramoA*).

```
PS C:\Users\arian\ProgettoGit> git checkout ramoB
Switched to branch 'ramoB'
PS C:\Users\arian\ProgettoGit> git rebase ramoA
Successfully rebased and updated refs/heads/ramoB.
```

### Risolvere merge conflicts

Quando si tenta di unire (con *merge* o *rebase*) rami che hanno apportato **modifiche diverse** nella stessa riga in uno **stesso file** del progetto, si incorre in dei *merge conflicts*. Se si esegue *git status*, viene segnalata la presenza dei conflitti e vengono indicati i file interessati.

Nell'esempio seguente, si prova a fondere un ramo *ramoB* in un ramo *ramoA*, ma entrambi hanno cambiato la prima riga di un file di testo «*testoA.txt*»:

```
PS C:\Users\arian\ProgettoGit> git checkout ramoA
Switched to branch 'ramoA'
PS C:\Users\arian\ProgettoGit> git merge ramoB
Auto-merging testoA.txt
CONFLICT (content): Merge conflict in testoA.txt
Automatic merge failed; fix conflicts and then commit the result.
PS C:\Users\arian\ProgettoGit> git status
On branch ramoA
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   testoA.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Per risolvere un merge conflict su un file nome\_file si può:

- forzare il *merge* a mantenere solo le modifiche apportate dal ramo selezionato (nell'esempio *ramoA*) usando *git checkout --ours* nome\_file, *git add* nome\_file e *git commit -m* "messaggio";
- forzare il *merge* a mantenere solo le modifiche apportate dal ramo che si sta fondendo (nell'esempio *ramoB*), usando *git checkout --theirs* nome\_file, *git add* nome\_file e *git commit -m* "messaggio";
- modificare manualmente il contenuto del file nome\_file, in modo da poter integrare i cambiamenti dai due rami. Quando si verifica un merge conflict,

il contenuto dei file interessati include entrambe le versioni dei rami coinvolti dal merging: Git delimita con «<<<<<<<<» e «=====» la versione del ramo selezionato e con «=====» e «>>>>>>>>» quella del ramo che si sta fondendo. Una volta scelte le modifiche da tenere ed eliminati i delimitatori, si può procedere con `git add nome_file` e `git commit -m "messaggio"` per completare il *merge*.

Nell'esempio, si risolve un merge conflict modificando manualmente il contenuto del file «testoA.txt»:

```
<<<<<<<< HEAD
modifica dal ramo A !
=====
modifica dal ramo B !
>>>>>>>> ramoB
```

Figura 22:

Contenuto del file «testoA.txt» dopo il tentativo di *merging* del ramo *ramoB* nel ramo *ramoA* (puntato da *HEAD*)

```
modifica dai rami A e B !
```

Figura 23:

Contenuto del file «testoA.txt» dopo una modifica manuale arbitraria eseguita per risolvere il *merge conflict*

```
PS C:\Users\arian\ProgettoGit> git add testoA.txt
PS C:\Users\arian\ProgettoGit> git commit -m "merge di ramoB in ramoA"
[ramoA 6163c3b] merge di ramoB in ramoA
```

## Lavorare in gruppo

I comandi finora discussi permettono di lavorare su un progetto in **locale**, cioè esclusivamente sul proprio computer. Tuttavia, a volte si desidera salvare una copia dei dati su un server per maggiore sicurezza, oppure si vuole collaborare con altre persone e condividere i file del progetto ponendoli su un repository **remoto**. Come fare?

### Connettersi a un repository remoto

Si definisce *remote repository* qualsiasi copia del progetto che, rispetto a quella correntemente considerata, sia ospitata in un «luogo» diverso, che può essere una zona di internet, come nel caso di un repository su GitHub, ma anche un'altra cartella del filesystem locale.

Il comando `git remote -v` permette di visualizzare i repository remoti (un nome breve e l'URL corrispondente) salvati per il progetto.

Per salvare esplicitamente il riferimento ad un repository remoto (che può essere *read-only* o *read/write* in base ai propri permessi su di esso) si può eseguire `git remote add nome url`: in questo modo si aggiunge il repository identificato dall'URL indicato e in seguito ci si potrà riferire ad esso semplicemente con il nome fornito.

```
PS C:\Users\arian\ProgettoGit> git remote add esempio https://github.com/arianna011/esempio-git
PS C:\Users\arian\ProgettoGit> git remote -v
esempio https://github.com/arianna011/esempio-git (fetch)
esempio https://github.com/arianna011/esempio-git (push)
```

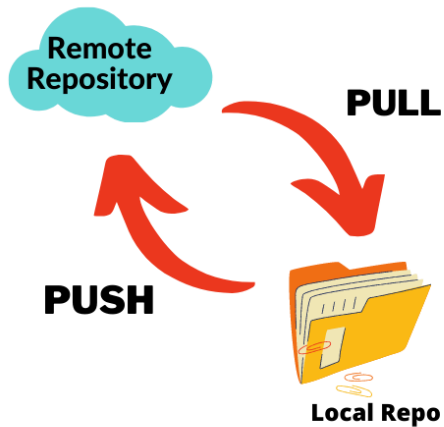
Per rimuovere un riferimento che non si intende più usare si può invece eseguire `git remote remove nome`, dove *nome* è il nome con cui si era salvato il repository remoto interessato.

Spesso si usa lavorare su una copia locale di un **repository remoto pre-esistente**: il comando `git clone url` **clona** il repository presente all'indirizzo url in una cartella nel proprio computer creata automaticamente con il nome del repository (alternativamente si può specificare la cartella dove posizionare la copia del repository usando `git clone url cartella`). Git salverà implicitamente il riferimento al repository remoto originale, denominandolo «*origin*».

```
PS C:\Users\arian\ProgettoGit> git clone https://github.com/arianna011/esempio-git
Cloning into 'esempio-git'...
warning: You appear to have cloned an empty repository.
PS C:\Users\arian\ProgettoGit> cd esempio-git
PS C:\Users\arian\ProgettoGit\esempio-git> git remote -v
origin https://github.com/arianna011/esempio-git (fetch)
origin https://github.com/arianna011/esempio-git (push)
```



## Fare pull e push



Per sincronizzare la **copia locale** di un progetto con le modifiche eseguite sulla sua **versione in remoto** o, viceversa, propagare i cambiamenti eseguiti in locale sulla copia remota, si possono eseguire le azioni opposte di *pull* («tirare», cioè portare anche in locale l'ultima versione remota del progetto) e *push* («spingere», cioè mandare anche in remoto l'ultima versione locale del progetto).

In aggiunta, esiste l'operazione di *fetch* («prendere»), che permette di visualizzare in locale le modifiche effettuate sulla versione remota del progetto, senza però sovrascrivere o interagire con i file della cartella locale di lavoro. Si può procedere così in modo cauto, decidendo solo dopo aver controllato le nuove modifiche se unirle o meno alla copia locale del progetto con un *merge*. Il *pull*, invece, esegue automaticamente la fusione delle modifiche importate dalla versione remota.

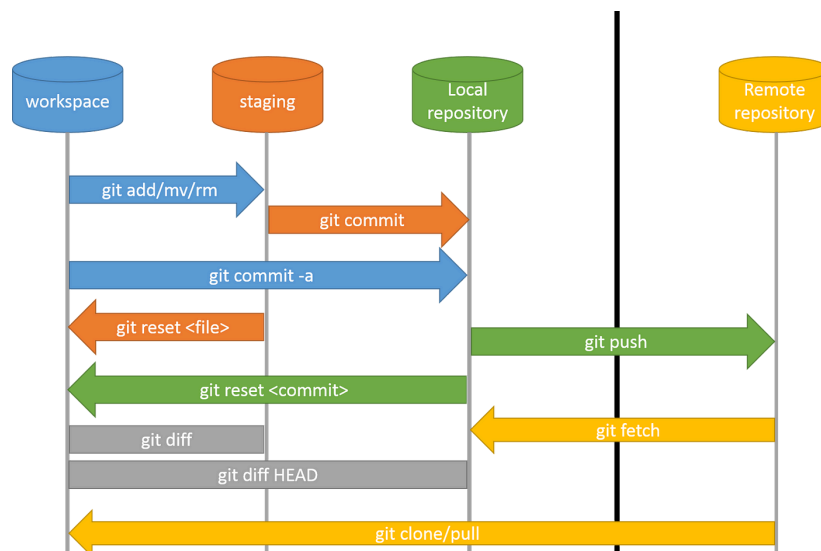
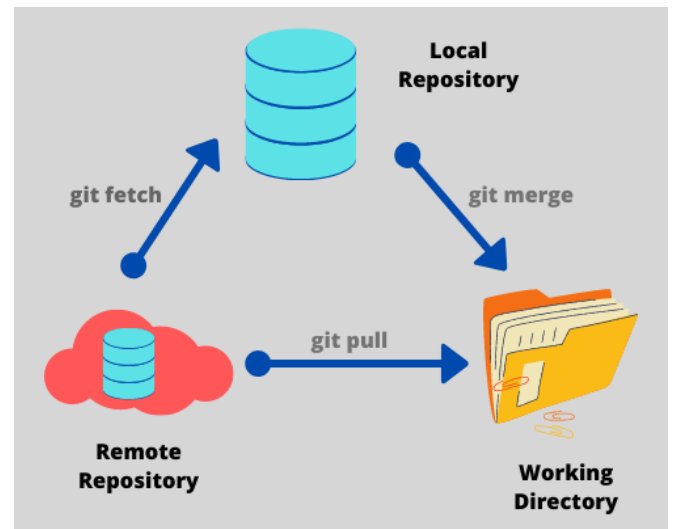


Figura 24:

Illustrazione del funzionamento dei comandi *push*, *fetch* e *pull*: *push* sincronizza i cambiamenti del repository locale (realizzati tramite commits) sul repository remoto; *fetch* prende i cambiamenti del repository remoto e li mette a disposizione nel repository locale, senza interagire con la cartella di lavoro («workspace»); *pull* fonde direttamente i cambiamenti del repository remoto con la cartella di lavoro locale

Il comando `git push nome_repository_remoto nome_ramo_locale` esegue il push del ramo locale `nome_ramo_locale` del progetto (nell'esempio «*ramoA*») sul repository remoto salvato con il riferimento `nome_repository_remoto` (nell'esempio «*esempio*»).

```
PS C:\Users\arian\progettoGit> git push esempio ramoA
Enumerating objects: 41, done.
Counting objects: 100% (41/41), done.
Delta compression using up to 8 threads
Compressing objects: 100% (29/29), done.
Writing objects: 100% (41/41), 3.65 KiB | 748.00 KiB/s, done.
Total 41 (delta 7), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (7/7), done.
To https://github.com/arianna011/esempio-git
 * [new branch]      ramoA -> ramoA
```

Si può anche digitare `git push --all nome_repository_remoto` per fare il push di tutti i rami locali del progetto sul repository remoto.

Il comando `git fetch nome_repository_remoto` esegue il fetch di tutti i rami del repository remoto con il nome fornito (alternativamente si può specificare un solo ramo da includere con `git fetch nome_repository_remoto nome_ramo_remoto`). Tali rami potranno essere visualizzati con `git branch -r` ed eventualmente essere uniti ai file locali di lavoro tramite merging. Nell'esempio seguente, il ramo «*nuova-feature*» del repository remoto «*esempio*» viene fuso nel ramo «*master*» locale:

```
PS C:\Users\arian\progettoGit> git fetch esempio
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (4/4), 1.86 KiB | 95.00 KiB/s, done.
From https://github.com/arianna011/esempio-git
 * [new branch]      master      -> esempio/master
 * [new branch]      nuova-feature -> esempio/nuova-feature
 * [new branch]      ramoA       -> esempio/ramoA
 * [new branch]      ramoB       -> esempio/ramoB
PS C:\Users\arian\progettoGit> git branch -r
esempio/master
esempio/nuova-feature
esempio/ramoA
esempio/ramoB
```

```
PS C:\Users\arian\progettoGit> git checkout master
Switched to branch 'master'
PS C:\Users\arian\progettoGit> git merge esempio/nuova-feature
Updating 78c230c..a6aec73
Fast-forward
 file-remoto.txt | 1 +
 testoA.txt      | 2 +-
 testoB.txt      | 2 +-
 testoC.txt      | 1 +
 4 files changed, 4 insertions(+), 2 deletions(-)
 create mode 100644 file-remoto.txt
```

Il comando `git pull nome_repository_remoto` incorpora automaticamente i cambiamenti eseguiti nel repository remoto `nome_repository_remoto` all'interno del ramo locale correntemente selezionato, eseguendo un fetch seguito da un merge automatico. Richiede quindi attenzione, in quanto si potrebbero sovrascrivere involontariamente alcune modifiche locali. Si può usare anche `git pull nome_repository_remoto nome_ramo_remoto` per includere solo le modifiche eseguite in remoto sul ramo `nome_ramo_remoto` (nell'esempio «*ramoA*»).

```

PS C:\Users\arian\progettoGit> git pull esempio ramoA
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 939 bytes | 62.00 KiB/s, done.
From https://github.com/arianna011/esempio-git
 * branch            ramoA      -> FETCH_HEAD
    eed4963..ddee743  ramoA      -> esempio/ramoA
Updating a6aec73..ddee743
Fast-forward
 file-remoto.txt | 2 + -
 1 file changed, 1 insertion(+), 1 deletion(-)

```

## Gestire un progetto con Git e GitHub

Come menzionato in precedenza, se **Git** è un software di controllo di versione per gestire la storia di un progetto, **GitHub** è una piattaforma per mantenere e condividere i file, permettendo all'utente di usare Git implicitamente tramite interfaccia grafica.

Di seguito vengono presentati alcuni passi pratici per l'utilizzo di Git e GitHub. Per eseguirli è necessario aver installato [Git](#) ed aver creato un account su [GitHub](#) (durante i passaggi sarà inoltre probabilmente richiesta l'autenticazione tramite email e password).

### Caricare un repository locale su GitHub

Poniamoci nella situazione in cui abbiamo già creato un repository locale Git da linea di comando e vogliamo farne l'**upload su GitHub**, così da gestirlo più facilmente ed eventualmente condividerlo con altri collaboratori.

Un modo per farlo prevede i seguenti passi:

1. creare un repository vuoto iniziale dal sito GitHub (Figura 25)
2. copiare l'url del repository appena creato (Figura 26)
3. aprire un terminale e posizionarsi nella cartella di lavoro contenente il repository Git locale che si vuole esportare su GitHub
4. salvare tramite Git il riferimento al repository remoto su GitHub come «*origin*» inserendo l'url copiato

```

PS C:\Users\arian\ProgettoGit> git remote add origin https://github.com/arianna011/nuovo-repository.git

```

5. fare un push delle modifiche sul ramo locale principale (nell'esempio «*master*») sul repository remoto *origin*

```

PS C:\Users\arian\ProgettoGit> git push origin master
Enumerating objects: 51, done.
Counting objects: 100% (51/51), done.
Delta compression using up to 8 threads
Compressing objects: 100% (36/36), done.
Writing objects: 100% (51/51), 7.18 KiB | 1.03 MiB/s, done.
Total 51 (delta 10), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (10/10), done.
To https://github.com/arianna011/nuovo-repository.git
 * [new branch]      master -> master

```

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (\*).

### Repository template

No template ▾

Start your repository with a template repository's contents.

Owner \*

arianna011 ▾

Repository name \*

/ nuovo-repository

✓ nuovo-repository is available.

Great repository names are short and memorable. Need inspiration? How about [crispy-computing-machine](#) ?

Description (optional)

☐ Public

Anyone on the internet can see this repository. You choose who can commit.

☒ Private

You choose who can see and commit to this repository.

Initialize this repository with:

☐ Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None ▾

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

ⓘ You are creating a private repository in your personal account.

Create repository

Figura 25:  
Schermata di creazione di un repository su GitHub

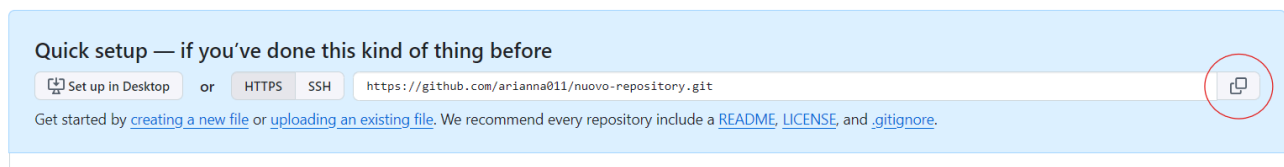


Figura 26:

Bottone per copiare l'url di un repository GitHub appena creato

Il repository contenente il progetto sarà ora visibile anche su GitHub (Figura 27).

## Scaricare un repository GitHub in locale

Se si desidera lavorare in locale a partire da codice trovato su GitHub, è possibile creare una **copia del repository online** sul proprio computer. Si può decidere di:

- scaricare l'intera cartella zip del repository GitHub considerato (Figura 28), per poi creare un repository Git locale nella cartella corretta (come visto nelle sezioni precedenti), in modo da ottenere un repository locale indipendente da quello remoto originale;
- fare un clone del repository GitHub tramite comando Git per avere una copia locale connessa a quella remota tramite il riferimento *origin*.

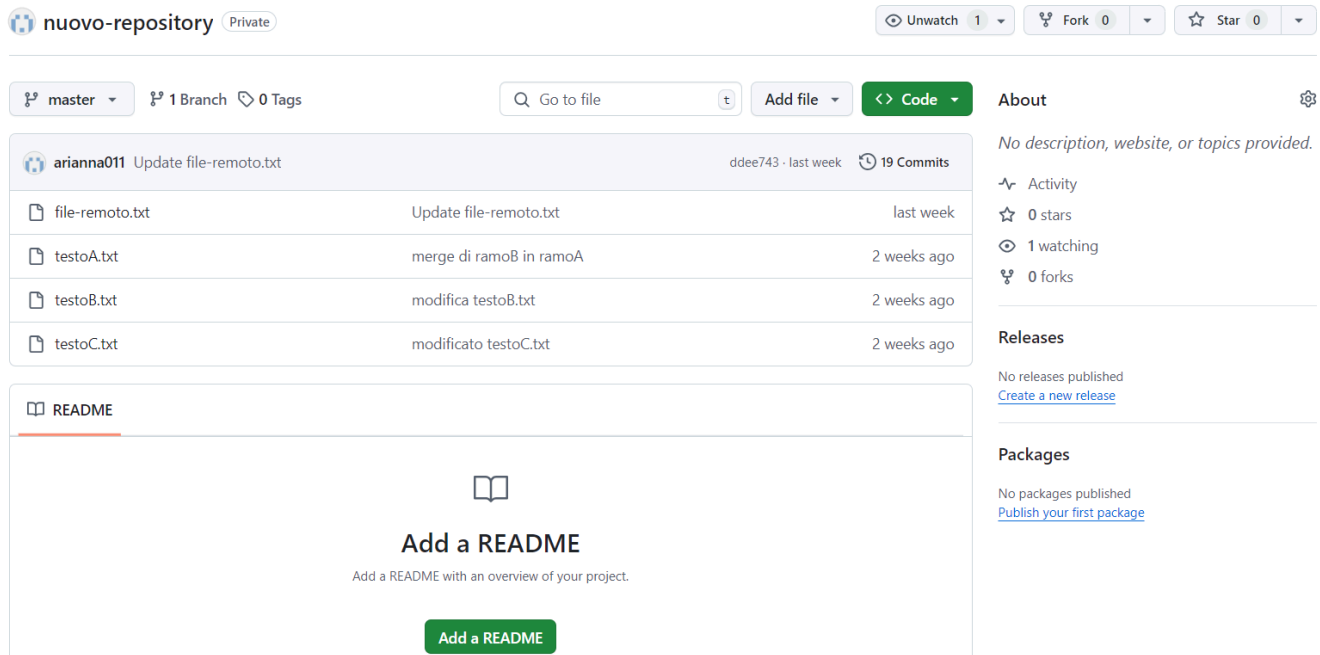


Figura 27:

Schermata del repository GitHub con le modifiche importate dalla cartella di lavoro locale

Se si sceglie l'opzione b) è bene sapere che:

- se si clona un repository GitHub pubblico di cui non si è contributori il repository sarà *read-only*, cioè non si potranno realizzare modifiche applicabili sul repository remoto;
- se si clona un repository privato personale verrà chiesto di inserire il proprio username e password (sarà possibile leggere e modificare il repository remoto).

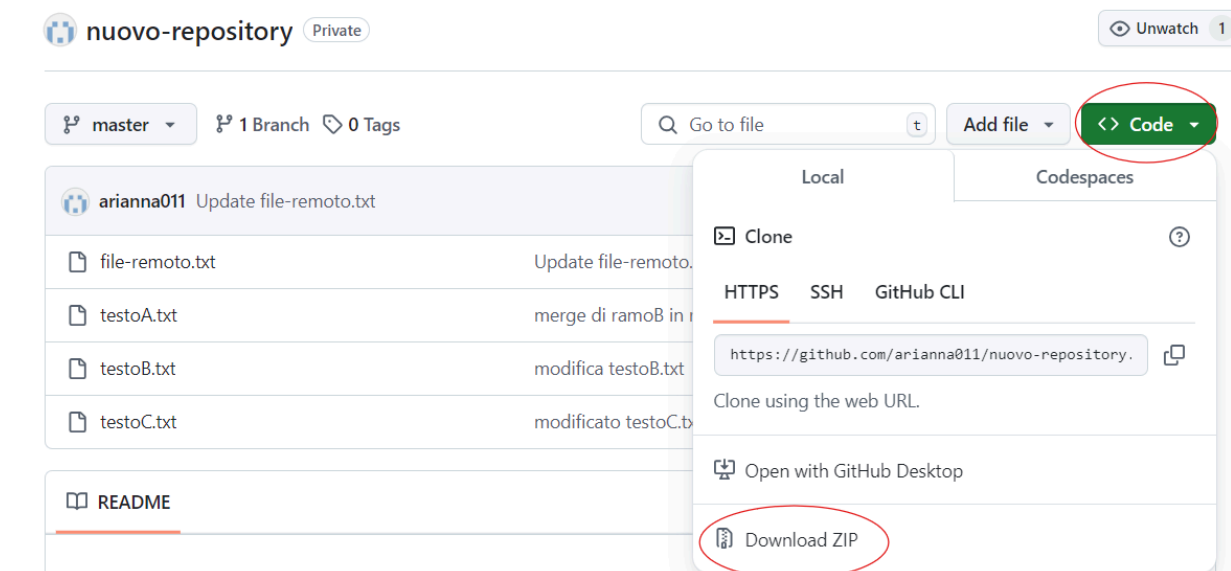


Figura 28:

Bottone per scaricare lo zip contenente tutti i file di un repository GitHub

Il comando `git clone url`, come visto in precedenza, permette di clonare il repository all'indirizzo url in una cartella locale creata automaticamente con

il nome del repository stesso. Il repository remoto originale sarà accessibile con il riferimento «*origin*» per eseguire operazioni di pull e push.

## Collaborare

Git e GitHub sono strumenti utili quando si vuole **condividere un progetto** con più collaboratori: si può sfruttare GitHub per mantenere online una versione comune dei file, di cui ognuno avrà una copia locale su cui sincronizzare le modifiche realizzate dagli altri, tramite *pull*, e applicare le proprie, da condividere tramite *push*.

Per poter lavorare sullo stesso repository GitHub, tutti i partecipanti al progetto dovranno avere l'accesso ad esso come collaboratori. E' possibile aggiungere collaboratori ad un repository personale invitando utenti GitHub tramite le impostazioni della pagina del repository stesso (Figura 29).

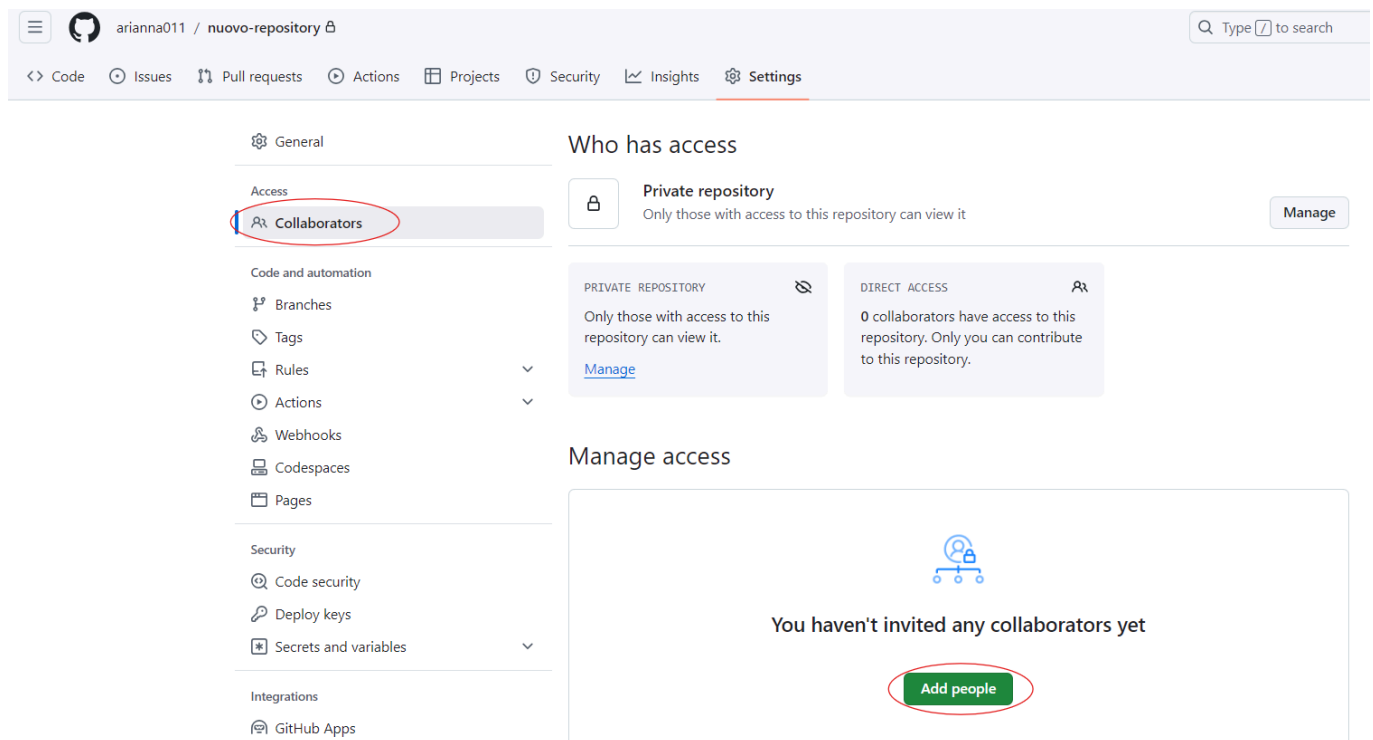


Figura 29:

Bottone per aggiungere un utente come collaboratore su un repository GitHub

## GitHub flow

Il GitHub flow è un procedimento di lavoro consigliato da GitHub per i team. Esso si basa sul *branching*: ogni collaboratore dovrebbe realizzare una nuova modifica su un ramo separato, in modo da tenere il ramo principale del progetto pulito e funzionante.

Ad esempio, se un collaboratore volesse sviluppare una certa feature del progetto, potrebbe creare un ramo *R* apposito e sperimentare liberamente su di esso. Una volta soddisfatto del risultato ottenuto, potrebbe fondere il ramo *R* nel ramo principale di lavoro, dove tutti i collaboratori sincronizzano le proprie modifiche. Prima di farlo, però, è bene avere l'approvazione degli altri membri del team. Per questo esiste un meccanismo di *pull requests*.

## Pull requests

Una **pull request** è una richiesta di revisione dei cambiamenti apportati su un ramo del progetto e di conseguente approvazione per la fusione nel ramo di lavoro principale. In un team, sarebbe bene avere un numero limitato di persone che si occupano di gestire le pull requests controllando i cambiamenti che richiedono di apportare e decidendo se autorizzare il *merge* o attendere un ulteriore miglioramento del codice.



Figura 30:  
Illustrazione del meccanismo di pull request

Quando, dunque, si è pronti ad unire i cambiamenti realizzati su un proprio ramo separato al ramo principale del progetto, si può creare una pull request da interfaccia grafica su GitHub (Figura 31, Figura 32) e attendere l'approvazione o eventualmente una richiesta di revisione del codice tramite commento testuale.

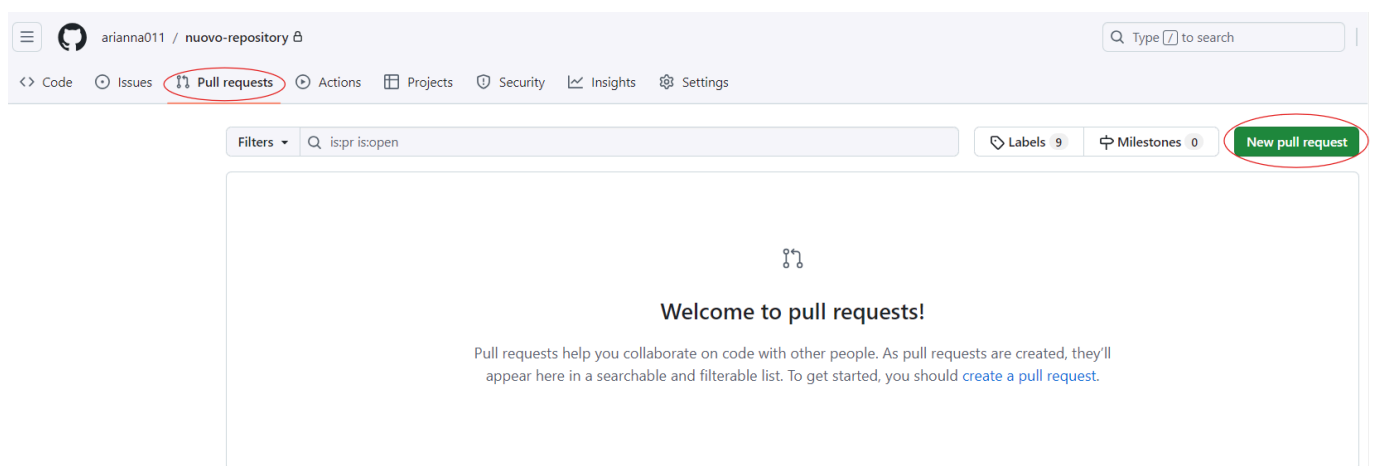


Figura 31:  
Bottone per aggiungere una nuova *pull request*

A questo punto, i «reviewers» designati o in generale chi decide di occuparsi della pull request (nella pratica può essere chiunque) può analizzarla e lasciare dei commenti testuali, oltre che decidere di chiuderla e realizzare il

Schermata con la discussione riguardo una *pull request* e il bottone per eseguire il merge (in questo esempio non ci sono *merge conflicts* con il ramo principale, in caso contrario sarebbe necessario risolverli prima di eseguire la fusione automatica)



## Forks

Se si vuole collaborare ad un repository GitHub su cui non si hanno permessi di scrittura o utilizzare del codice *open source* come punto di partenza per un proprio progetto, è possibile sfruttare il meccanismo del *fork*.

Tale operazione consente, infatti, di creare sul proprio account una copia personale di un repository GitHub, che sarà indipendente da esso: i push/pull non influenzeranno in alcun modo il repository originale, detto ***upstream repository***.

Si può quindi procedere clonando il repository risultante dal fork (chiamato esso stesso ***fork***) per ottenere i file del progetto in locale (il fork agisce solo su GitHub), per lavorarci autonomamente oppure proporre modifiche da applicare all'*upstream repository* tramite *pull requests* (Figura 34).

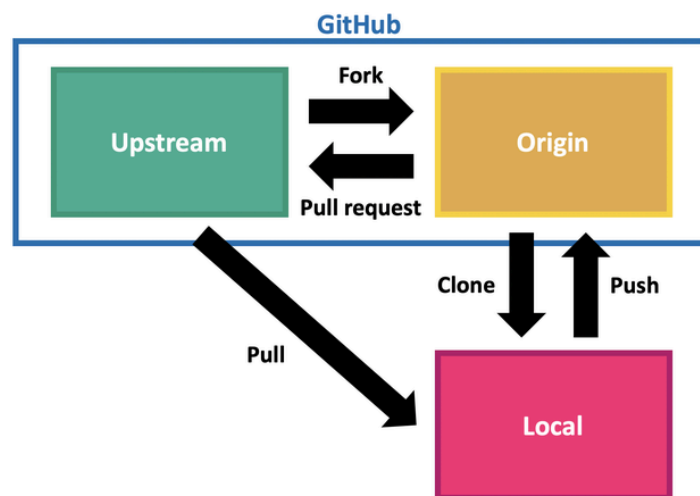


Figura 34:

Illustrazione dei repository coinvolti in un'operazione di *fork* e *clone*: l'*upstream repository* è il repository originale di cui si ottiene una copia su GitHub tramite *fork*. Tale copia può essere clonata per ottenere un repository *locale*, per il quale essa sarà l'origine remota (*origin*). Si può aggiungere un riferimento remoto all'*upstream repository* nel repository locale per sincronizzare le modifiche provenienti da esso tramite *pull*. Le operazioni di *push* potranno invece avere effetto solo sul repository *origin* personale, dato che non si hanno permessi di scrittura sull'*upstream*. Per provare a realizzare modifiche sull'*upstream repository* si possono creare *pull requests* che gli autori dovranno approvare

Per realizzare un *fork* basta cliccare sull'apposito bottone nella pagina GitHub del repository che si vuole «forkare» (Figura 35) e configurare alcuni parametri iniziali come titolo e descrizione del nuovo repository (Figura 36).

Per creare una *pull request* al fine di incorporare i propri cambiamenti nel repository originale si può, invece, procedere in modo simile a quanto visto nella sezione precedente: aprire la pagina GitHub dell'*upstream repository*, cliccare sul bottone «Pull requests» e poi su «New pull request», scegliere i rami di cui si vuole proporre la fusione (un ramo dell'*upstream repository* e un ramo del repository forkato) e completare i campi richiesti per la creazione

della richiesta, che sarà visibile agli autori dell'upstream repository e disponibile per eventuale approvazione.

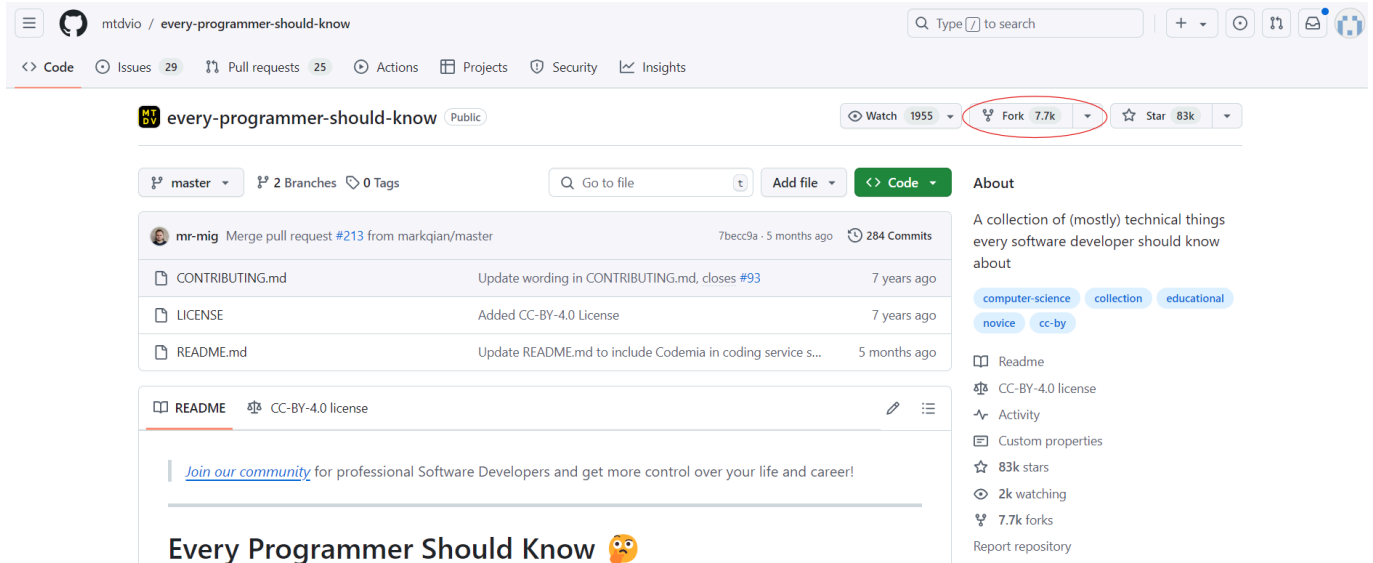



Figura 35:  
Bottone per ottenere un fork del repository GitHub selezionato

## Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

Required fields are marked with an asterisk (\*).

Owner \*

 arianna011

Repository name \*

/ every-programmer-shouldk

✔ every-programmer-should-know is available.


By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

A collection of (mostly) technical things every software developer should know about

☒ Copy the **master** branch only

Contribute back to mtdvio/every-programmer-should-know by adding your own branch. [Learn more.](#)

 You are creating a fork in your personal account.

Create fork

Figura 36:  
Schermata per configurare un fork da creare

## Link utili

Vengono qui riportate alcune risorse (utilizzate in parte come riferimento per questa guida) che, insieme a quelle linkate in alcune didascalie di immagini, sono utili per approfondire ulteriormente Git e GitHub o configurare il loro utilizzo pratico (nota: la maggior parte del materiale è in lingua inglese).

- **Documentazione ufficiale di Git**, utile per capire il funzionamento dei suoi comandi e scoprire le varie opzioni disponibili: <https://git-scm.com/docs>.
- **Documentazione ufficiale di GitHub**, per comprendere i meccanismi che offre: <https://docs.github.com/en>.
- **Lezione su Git** da *The Missing Semester of Your CS Education* (video lezione con appunti testuali sul controllo di versione, la struttura dati e alcuni comandi di Git): <https://missing.csail.mit.edu/2020/version-control/>.
- **Learn Git Branching**, un gioco su browser a livelli (anche in italiano) per imparare a gestire il modello a grafo di Git tramite operazioni di branching, merging, rebasing, ecc.: [https://learngitbranching.js.org/?locale=it\\_IT](https://learngitbranching.js.org/?locale=it_IT).
- **How to use Eclipse with GitHub**, una guida su come configurare l'IDE Eclipse per poter usare GitHub direttamente dalla sua interfaccia grafica: <https://github.com/maxkratz/How-to-Eclipse-with-Github>.
- **Introduction to Git in VS Code**, una guida per collegare Visual Studio Code con Git e GitHub: <https://code.visualstudio.com/docs/sourcecontrol/intro-to-git>.