



git

introduzione

cos'è git

Uno strumento per tracciare cambiamenti fatti nel tempo, meccanismo noto come **version control**.

- esamina lo stato dei tuoi progetti al più “**antico**” punto nel tempo
- mostra le differenze tra vari stati del progetto
- divide lo sviluppo del progetto in più linee indipendenti, **branches**
- periodicamente ricombina i **branches** in un processo chiamato **merging**
 - permette a più persone di lavorare simultaneamente, condividendo e combinando il loro lavoro a seconda delle necessità



cosa non è!

Git è un sistema distribuito di controllo di versione “free as in beer” e open source , pensato per gestire file (piccoli o grandi) con velocità ed efficienza (<https://git-scm.com>).

GIT NON È GITLAB

Git è lo strumento

GitLab è il servizio che ospita i tuoi progetti gestiti con git

GitLab ti fornisce una repository cloud con cui sincronizzare i tuoi progetti git, fornisce anche altri servizi tra cui CI/CD, wiki, ticketing



come funziona?

Un progetto git è una **repository** che contiene l'intera storia del progetto dalle sue origini.

Una repository è un insieme di **singoli snapshot** del progetto chiamati **commits**.

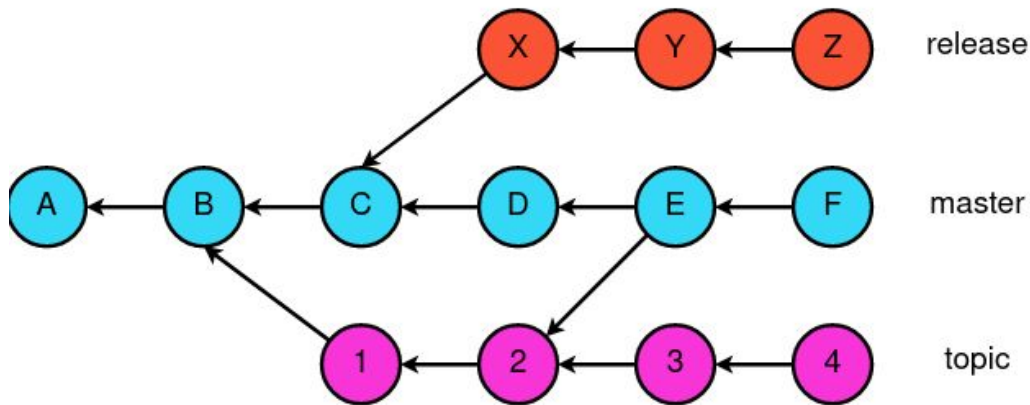
La struttura dei file/cartelle che rappresentano l'intero stato del progetto è detta **tree**

Un commit rappresenta uno stato del tree grazie ad **author**, **committer**, **commit message** e lista di uno o più **parent commits**



come funziona?

Il set di tutti i commit di una repository connessi da linee indicanti i loro parent commit formano una figura chiamata **repository commit graph** (un grafo aciclico diretto - o orientato -)



release

lettere e numeri
rappresentano i **commit**

master

commit senza padre sono **root commit**

topic

se ho un commit ha più di un
padre è detto **marge commit**



le etichette a destra del grafico sono chiamati **branches**
l'ultimo commit di ogni branch è detto **tip** del branch

perchè?

due contesti principali:

- **privato**: commit frequenti, tanti branch, per avere la libertà di poter sperimentare senza la preoccupazione di dover recuperare uno stato precedente → Comandi principali: **add, commit**
- **pubblico**: condivisione di uno stato stabile/finito come somma di uno o più commit locali → Comandi principali: **fetch, pull, push**

questa distinzione riflette il meccanismo base di git
ovvero la separazione tra **commit** e **push**, tra **lavoro locale** e **condivisione/pubblicazione**



git object store

Il git object store è un database che contiene 4 tipi di elementi: blobs, trees, commits, tags

blob: un “pezzo” di dati opaco, ovvero **un insieme di byte senza una precisa struttura interna** (per quel che riguarda git)

ogni versione di un file in git è rappresentata per intero (non come differenza dal precedente): più spazio occupato, più veloce, più sicuro

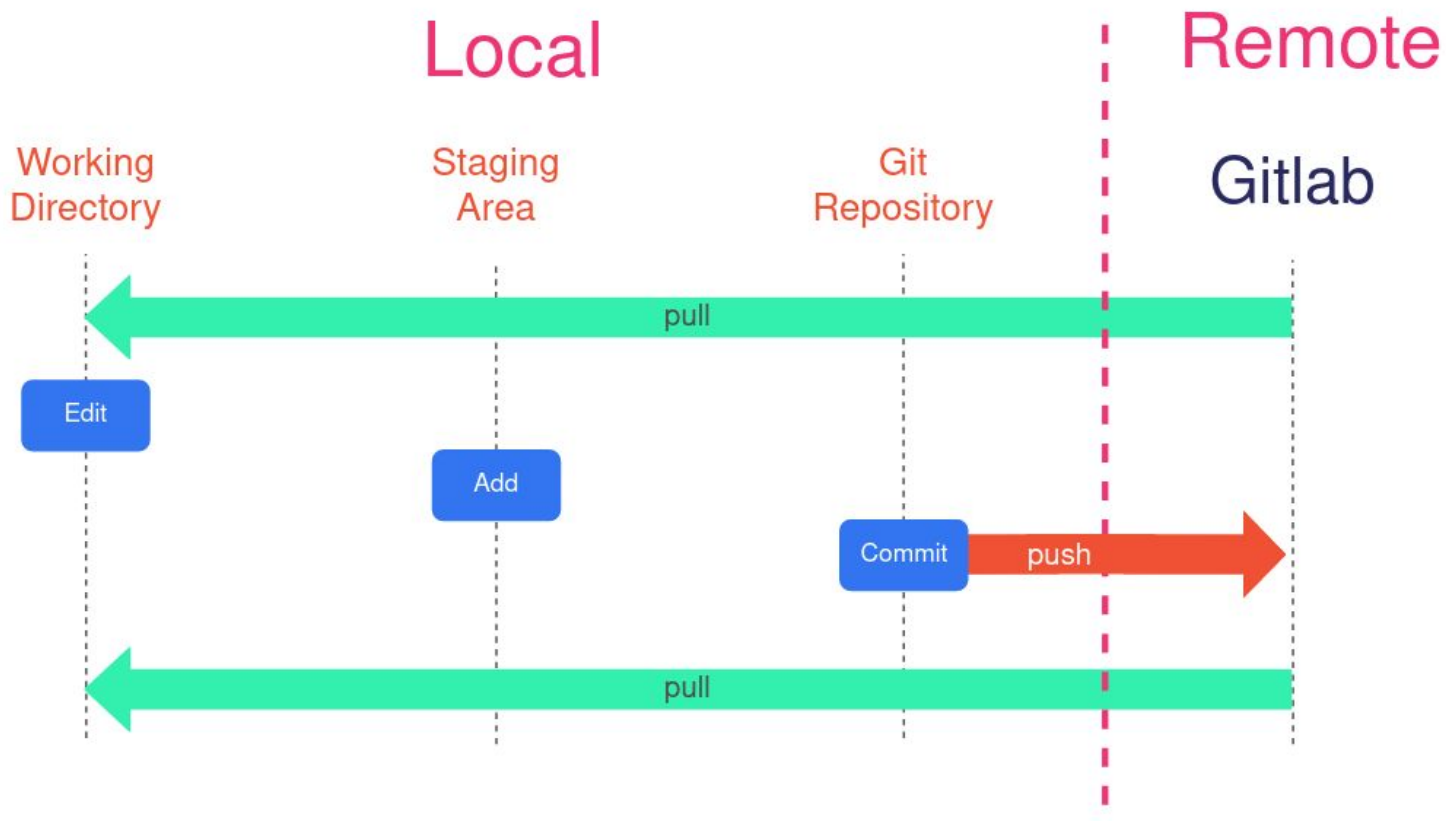
un **tree** in realtà è un solo livello della repository, **rappresenta un punto nel tempo della repository**
contiene una lista di elementi (file con dati che git traccia e puntatore ad un blob)

commit: snapshot dello stato del progetto. Contiene un puntatore al tree di root.

tag: etichetta human-readable che punta a un particolare commit.
Spesso usata per indicare una versione rilasciata del progetto.



git workflow



git object store

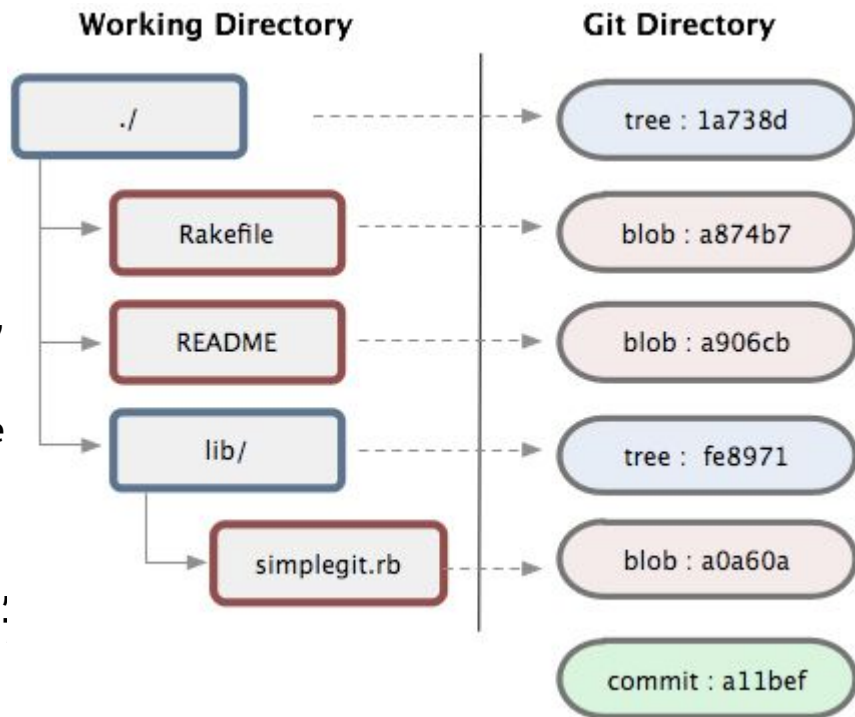
Gli oggetti git sono identificati tramite un **hash crittografico SHA1 (immutabili)**.

Gli oggetti si riferenziano a vicenda. Dunque la modifica ad un file implica la creazione di nuovi oggetti a cascata:

- un nuovo blob
- un nuovo tree contenente il blob
- eventuale nuovo tree contenente il tree modificato, ricorsivamente

Il meccanismo di hash e di riferimenti incrociati garantisce l'integrità del repository.

L'hash è composto di **40 caratteri**, ma, a meno di rarissime ambiguità, nei comandi basta digitare **i primi caratteri** per riferirsi all'oggetto.



install

installazione

Installazione di git →

Linux → [apt|yum] install git

MacOs e Windows → <https://git-scm.com/downloads>

documentazione → <https://git-scm.com/doc>

Come controllare se ha funzionato?

aprire un terminale e digitare: **git help**

output:

```
> git --help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
      [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
      [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
      [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
      [--super-prefix=<path>] [--config-env=<name>=<envvar>]
      <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
  clone                Clone a repository into a new directory
  init                 Create an empty Git repository or reinitialize an existing one
```



git config

Prima di utilizzare **git** è necessario configurare alcuni parametri base, per farlo utilizziamo il comando:

git config --[local|global|system] parameter value

- **--local**: configurazione valida per la repository corrente
- **--global**: configurazione valida ovunque per l'utente corrente
- **--system**: configurazione valida system-wide per tutti gli utenti



git config - parametri -

I parametri più comuni di git config sono: (in case of error **--unset**)

- **init.defaultBranch**: change "master" in "main"

```
> git config --global init.defaultBranch main
```

- **user.name**: nome utente applicato ai commit

```
> git config --global user.name "Francesco Faenza"
```

- **user.email**: configurazione valida ovunque per l'utente corrente

```
> git config --global user.mail "frfaenza@unimore.it"
```

- **--list**: mostra le configurazioni correnti

```
user.email=frfaenza@unimore.it
user.name=Francesco Faenza
user.signingkey=7537DAA18CAF268F
core.editor=vim
core.excludesfile=/home/cicciodev/.gitignore
commit.gpgsign=true
init.defaultbranch=main
(END)
```



init

git init

Per poter utilizzare **git** all'interno di un repository (che eventualmente contiene il nostro progetto), dobbiamo inizializzare git:

git init [directory]

Crea la *directory* se specificata e crea una directory chiamata **.git** all'interno della *directory* creata se specificata o di quella corrente.

.git conterrà il working tree cioè le copie dei file e directory che sono state incluse nel controllo di versione



commit

cos'è?

il commit è la fondamentale unità di cambiamento, è composto da:

- un puntatore ad un tree
- informazioni ausiliari (author e committer)
- una lista di 0 o più oggetti commit: parent commits

Un commit è **immutabile**: cambiarne i contenuti ne cambierebbe l'hash identificativo.

almeno un commit nella repository non ha parent (root commit)
si possono introdurre commit senza genitori, orfani (**git checkout --orphan**)
si può firmare il proprio commit con una chiave GPG (**--gpg-sign[=keyid]**)



tutti gli oggetti "committati" vivono in **.git/objects**

git status/add

Per avere un'idea della situazione della mia cartella: **git status**

```
> git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README.md

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

riporta il mio branch attuale, i commit precedenti, untracked files e cambiamenti da “committare”

Per aggiungere file alla mia staging area:

git add [<pathspec>...]



git commit/diff

Per spostare un oggetto nella mia git repository devo eseguire il comando:

```
git commit [-a | --interactive | --patch]
           [-F <file> | -m <msg>]
           [<pathspec>...]
```

Convenzionalmente prima eseguo git add per aggiungere i file voluti alla staging area (invece di effettuare un -a) e solitamente aggiungo un messaggio con il parametro -m

```
git commit -m "a useful commit message"
```



Molto utile verificare le differenze tra staged e git repository

```
git diff [<path> <path>]
```

un buon commit?

il **message** aiuta a definire un buon commit, non dovrebbe essere più lungo di 50-60 caratteri, se voglio aggiungere ulteriore descrizione posso lasciare una riga bianca e descrivere ulteriormente"

```
> git commit -m "added README.md file" -m "added README.md but it is empty or almost empty  
dquote> you should remember to integrate it constantly  
dquote> as committ goes"
```

Nell'esempio il primo -m è il messaggio del commit, il secondo è una multi-line description

due domande:

- riesco a descrivere tutto il commit in meno di 60 char?
- il messaggio descrive completamente il contenuto del commit?

se no -> dividi in più commit



un buon commit?

<https://xkcd.com/1296/>

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT
MESSAGES GET LESS AND LESS INFORMATIVE.



remove object

Per rimuovere un oggetto dalla git repo:

```
git rm [<pathspec>...]
```

Oppure rimuovo l'oggetto dal sistema e poi lo aggiungo ai file da tracciare nel commit con git add

```
rm <pathspec>  
git add <pathspec>
```

ovviamente devo effettuare un commit a seguire



```
> git status  
On branch main  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstage)  
    deleted:    unwanted.log
```

rename object

Per rinominare un object dalla git repo:

```
git mv [<oldpathspec> <newpathspec> ]
```

Oppure rinomino l'oggetto dal sistema e poi lo aggiungo ai file da tracciare nel commit con git add

```
mv <oldpathspec> <newpathspec>  
git add <newpathspec>
```

ovviamente devo effettuare un commit a seguire



```
> git status  
On branch main  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstage)  
    renamed:    base.html -> index.html
```


restore object

git restore permette di ripristinare i file modificati, se non è ancora stato fatto il commit.

Se si modifica un file, il comando **git status** suggerisce come procedere: add o restore

```
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
```

Con **git restore** verranno scartate le modifiche.

⚠ *Attenzione! Possibile perdita di dati!* ⚠



```
$ git restore base.html
$ git status
On branch main
nothing to commit, working tree clean
```

git revert

A volta capita di voler annullare un commit, ad esempio perché ha introdotto dei bug nel progetto. Il comando `git revert <commit>` produce un nuovo commit contenente le operazioni opposte a quelle del commit target (`git hist` mostrerebbe il commit ed il revert)

1. Si esegue il commit "errato"

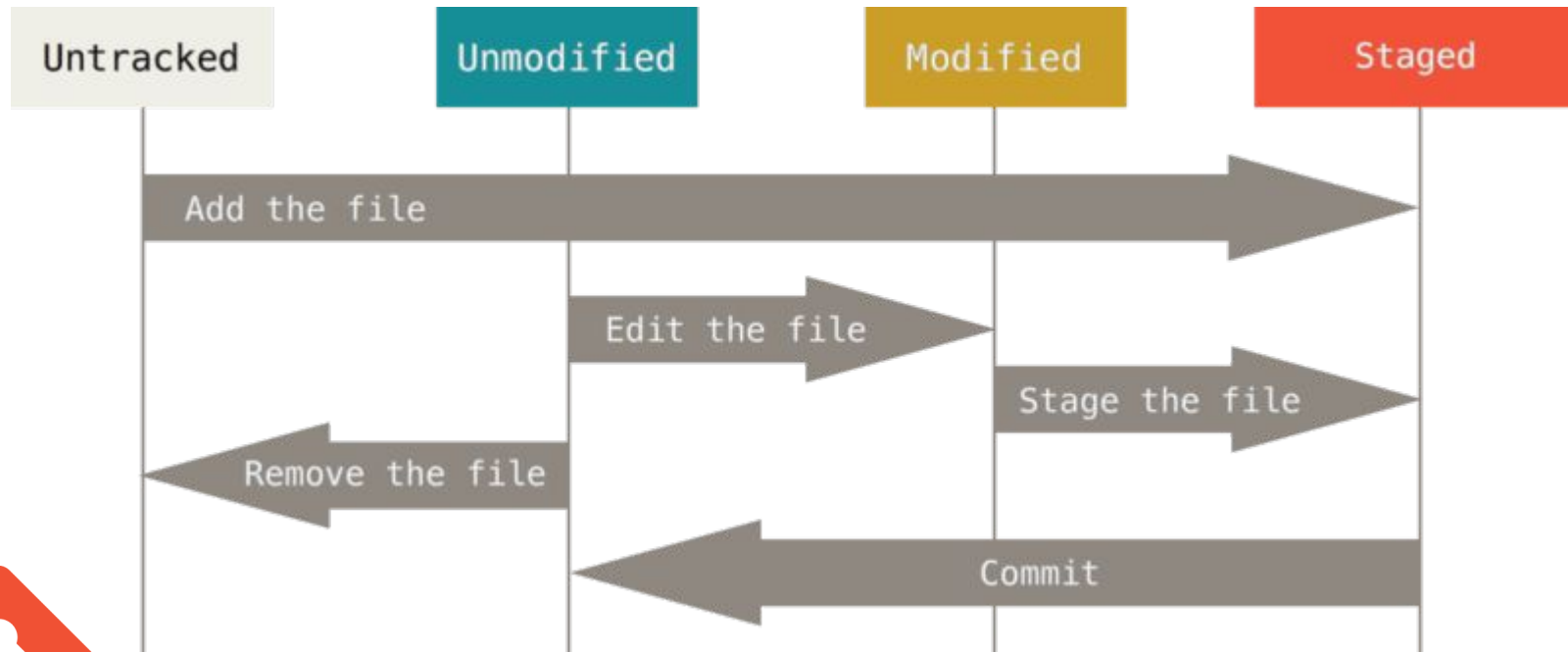
```
$ git add base.html  
$ git commit -m "Useless commit ahead!"  
[main 4576983] Useless commit ahead!  
1 file changed, 1 insertion(+)
```

2. Si ripristina lo stato precedente



```
$ git revert 4576983  
[main 42bf9e7] Revert "Useless commit ahead!"  
1 file changed, 1 deletion(-)
```

git add e commit



comandi utili

Variabile che mantiene il riferimento all'ultimo commit

HEAD

Reimpostare lo stato di un file o della repo all'ultimo stato della git repository (ovvero l'ultimo stato "committato")

git reset [--soft|--hard] [HEAD <path>]

Oppure stesso comando ma spostato solamente il mio indice HEAD

git checkout <path>

differenza tra reset e checkout:

<https://stackoverflow.com/questions/3639342/whats-the-difference-between-git-reset-and-git-checkout#answers>



Cambia il messaggio dell'ultimo commit

git commit --amend

.gitignore

cos'è?

è un file che contiene la lista di oggetti da ignorare nella directory, generalmente bytecode, file autogenerati, etc.

git guarda in 3 punti per capire cosa ignorare:

- **.gitignore** file nella root della working dir
- **.git/info/exclude** (è parte della repository configuration e NON del content)
- il file eventualmente specificato nella variabile **core.excludesfile**



esempio

contenuto tipico di un gitignore per python/django/pycharm

```
.idea  
*.pyc  
__pycache__
```

```
vim .gitignore  
git add .gitignore  
git commit -m "added .gitignore"
```



ignore patterns

Ignora un file specifico in una subdirectory

`conf/config.h`

Ignora un file nella root (non ./ ma /) directory

`/automatic-script.sh`

Tutti i pattern senza slash (/) si applicano ovunque:

Ignora estensioni *.pyc e *.pyo

`*.pyc`

`*.pyo`

Non ignorare un file specifico

`!my.pyc`

Ignora cartella ovunque sia

`__pycache__`



history

git log

Per visualizzare la storia della mia git repository:
(mi mostra HASH univoco identificante ogni singolo commit)

git log [--graph] [--oneline]

```

commit 083ce89c8c0d08f4cd06afbca1bae1809ffb391d (HEAD -> main)
Author: Francesco Faenza <dev@francescofaenza.it>
Date: Tue Mar 23 08:40:51 2021 +0100

    added .gitignore

commit 2f8cef7dd6948b739b3f64aeac07b7091aa84bf5
Author: Francesco Faenza <dev@francescofaenza.it>
Date: Tue Mar 23 08:27:55 2021 +0100

    added README.md file
(END)

```



remote

going remote

Creiamo un progetto su gitlab

Possiamo clonare la repo

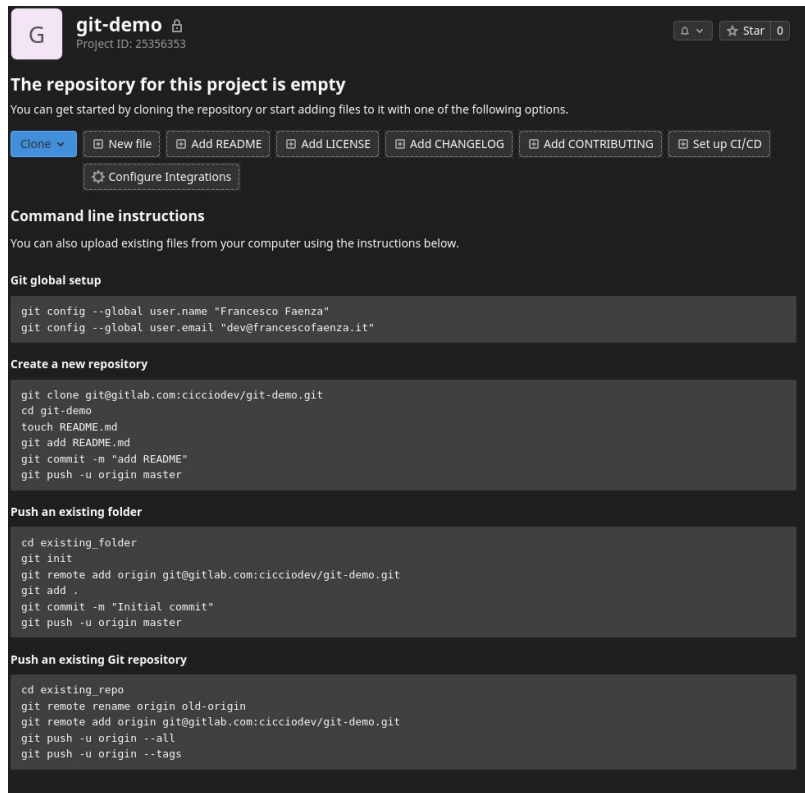
`git clone <ssh-repo-address>`

Oppure aggiungere/modificare un remote nella nostra local git repository

`git remote add origin <ssh-repo-address>`

Se ho già una remote, la rinomino prima di aggiungere la nuova

`git rename origin origin-old`



push/pull

push

Il push prova ad effettuare un update della repository remota con lo stato della repository locale, per far ciò è necessario conoscere la storia completa della remote repository:

`git fetch origin`

Se dovesse essere necessario un pull, fetch ci informerà della cosa, ora posso pushare

`git push -u origin main`

al primo push segnalo a che l'upstream è origin con branch main
d'ora in poi posso semplicemente fare



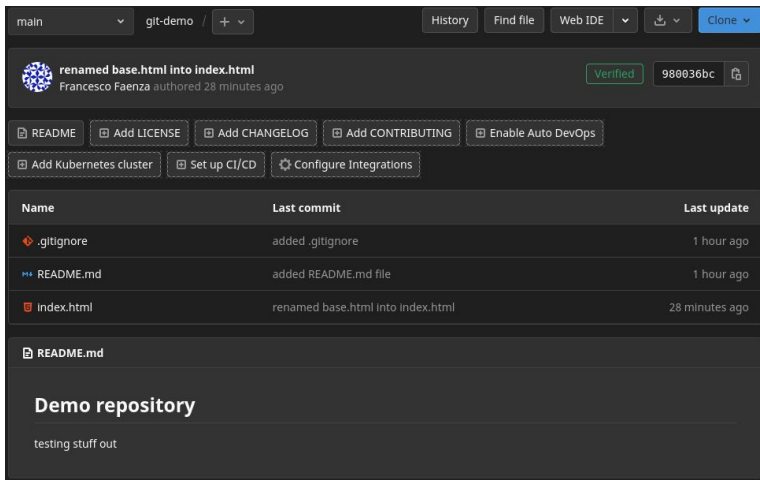
`git push`
`git push origin <branch>`

push - risultato -

risultato del primo push, notare l'ultima riga:

```
> git push -u origin main
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 16 threads
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), 5.25 KiB | 5.25 MiB/s, done.
Total 15 (delta 3), reused 0 (delta 0), pack-reused 0
To gitlab.com:cicciodev/git-demo.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Risultato nella repo online:

A screenshot of a GitLab repository page for 'git-demo'. The page shows a commit titled 'renamed base.html into index.html' by Francesco Faenza, made 28 minutes ago. Below the commit details, there are buttons for adding various files and configurations. A table lists the repository's files: .gitignore, README.md, and index.html, along with their last commit and update times. At the bottom, there is a section for the README.md file, which contains the text 'Demo repository' and 'testing stuff out'.

Name	Last commit	Last update
.gitignore	added .gitignore	1 hour ago
README.md	added README.md file	1 hour ago
index.html	renamed base.html into index.html	28 minutes ago



pull

Il pull esegue comunque un git fetch, per aggiornare il tracking locale della remote repository, ed ottiene tutti i nuovi oggetti necessari, blobs, commits, trees, etc.

Poi prova ad aggiornare la directory locale per “eguagliare” la repository remota

```
git pull  
git pull origin main  
git pull origin <branch>
```

È sempre bene “pullare” le modifiche da remoto prima di pushare le proprie

```
git pull  
git push
```



Di solito quando si lavora in team non è mai così semplice

branch

cos'è?

Un branch è un puntatore ad un commit come ref, meglio, tutti i punti raggiungibili in un commit graph dal named commit, “tip” of the branch. HEAD, ad esempio, è un ref simbolico al branch corrente (git symbolic-ref HEAD)

git branch

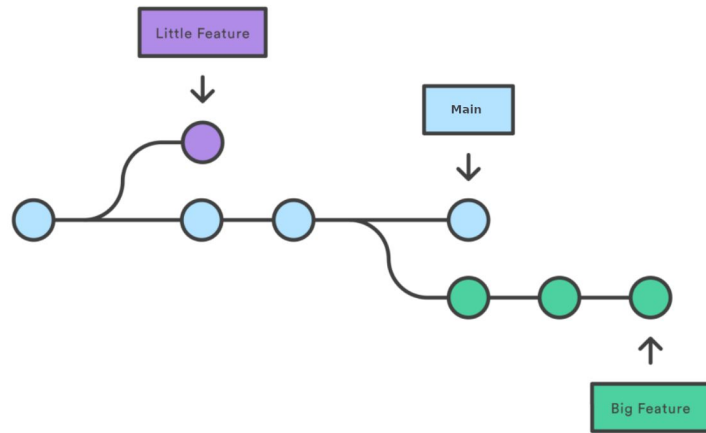
Restituisce l'elenco dei branch della mia repo e evidenzia quello corrente (*)

Create a new branch:

`git checkout -b <new-branch-name>`

Switch to a specific branch:

`git checkout <new-branch-name>`



come utilizzarli

Create a new branch:

`git checkout -b branch-name`

Switch to a specific branch:

`git checkout branch-name`

Rename a branch:

`git checkout -m old new`

Delete a branch: (*)

`git checkout -d branch-name`

Quando posso cancellare un branch?
Cancellando un branch cancello tutti i commit dalla tip di quel branch in poi
Il consiglio generale è di farlo solo in due casi:

- il branch viene “mergiato” in un altro e non ha più senso di esistere
- il branch è solo locale, quindi le mie azioni non impattano gli altri

<https://www.atlassian.com/git/tutorials/using-branches>



merge

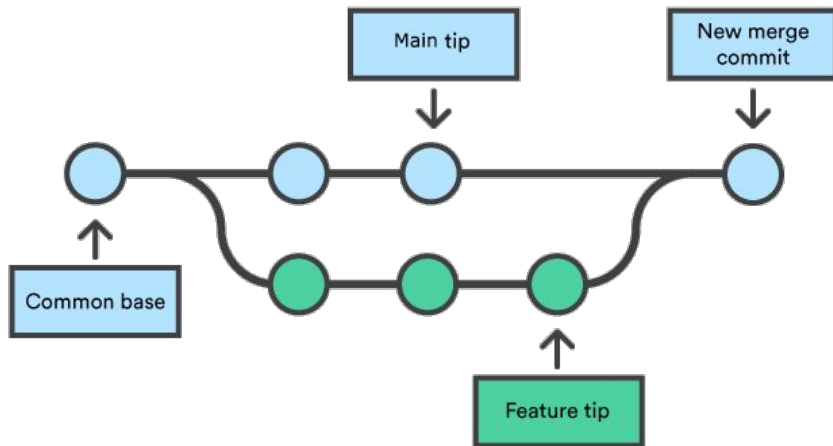
cos'è?

Una collaborazione efficace con git avviene se siamo in grado di modificare in maniera contemporanea la stessa repo, il merge ci permette di lavorare su branch separati per poi ricongiungersi in un “main” branch.

Il merge è il processo che combina più branch in un singolo commit che è l'insieme di quei branch

git merge branch-name

Fare questa operazione da riga di comando è **HARDCORE**, o meglio dovete sapere cosa state facendo



conflicts

In caso di conflicts git modifica il file in modo da riportare le differenze tra HEAD (branch corrente) e branch che ho deciso di mergiare

```

<<<< HEAD
<body>
<h1>Index</h1>
=====
<h1>Title modified</h1>
>>>> origin/dev
</body>

```

come leggere?

<<<< HEAD -> indica contenuto del branch corrente fino al divisorio

===== -> divisorio

>>>> origin/dev -> indica contenuto branch in ricezione



conflicts

Modifico il file in conflitto in modo da avere la versione corretta e committo

Molto più comodo utilizzare l'IDE (pycharm ha integrate funzioni di merge) per merge locali

Utilizziamo il nostro remoto host per merge remoti (Gitlab)



git branch & switch

git branch mostra una lista di tutti i branch – inizialmente solo main. Viene evidenziato anche **il branch attivo (HEAD)**, quello **a cui verranno aggiunti nuovi commit**.

git branch <nome> crea un nuovo branch, e **git switch <nome>** permette di renderlo attivo. I due comandi possono essere combinati con **git switch -c <nome>**

1. Si passa a un nuovo branch `$ git switch -c my-feature`
2. Il comando **git branch** conferma che esistono due branch, e my-feature è attivo.

```
$ git branch  
  
main  
* my-feature
```

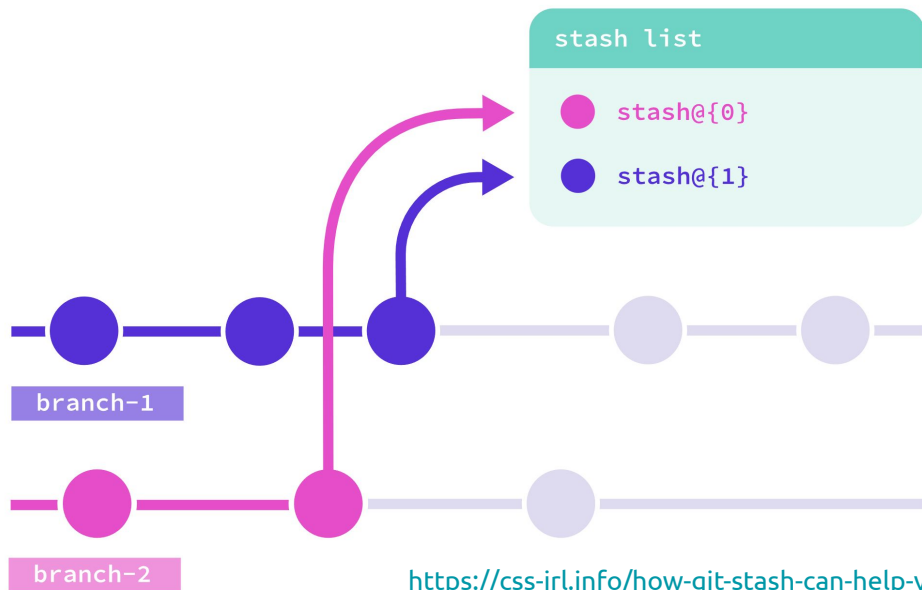


*N.B.: Anche **git status** mostra il branch attivo.*

git stash

Il comando `git stash` permette di “**mettere da parte** per dopo” le modifiche fatte al codice, senza farne il commit. L'utilizzo ideale si ha quando si viene interrotti nel mezzo di alcune modifiche, ed è necessario pulire il progetto o cambiare branch.

Con `git stash list` si visualizzano tutte le modifiche “messe da parte” con git stash.



remote

remotes

Git nasce come **sistema distribuito**, permette di **sincronizzare** il proprio repository con altri repo detti “**remotes**”.

Il repository remoto può trovarsi su un servizio di hosting come Github o Gitlab, su un computer aziendale o semplicemente in un'altra cartella.

L'istruzione `git remote add <nome repo> <indirizzo repo>` consente di aggiungere repository remoti. Il nome di default del remote principale è **origin**.

Per visualizzare i **remotes** già collegati si utilizza il comando `git remote -v`.



```
$ git remote -v  
origin  git@gitlab.com:frfaenza/my-project.git (fetch)  
origin  git@gitlab.com:frfaenza/my-project.git (push)
```

git clone

Con `git clone <remote> <cartella>` si può clonare un repo git, con tutta la sua storia e i suoi branch.

1. Si clona il progetto precedente “tutorial” in una nuova cartella, fornendo il path relativo o assoluto. Il nuovo repo avrà già un remote di nome `origin`:

```
$ git clone https://gitlab.com/cicciodev/cloudedgecomputing cec
Cloning into 'cec'...
done.
```



Anche nel caso di un progetto nuovo, i servizi di hosting di solito inizializzano subito il repo remoto e si può procedere direttamente con la clonazione.

remotes

La **sincronizzazione** con i remotes avviene solamente quando esplicitamente richiesta. Il repository tiene traccia del proprio stato e di quello dei remotes **indipendentemente**.

In git i comandi legati alla sincronizzazione sono sempre **monodirezionali**:

- con **git fetch** o **git pull** il repo locale **riceve** dal remote tutti i branch e i commit
- con **git push** si **invia** al remote il branch e il commit corrente e tutti i suoi antenati

I branch del remote prendono il nome **<nome remote>/<nome branch>** così da disambiguare il proprio branch **main** dal branch remoto **origin/main**.

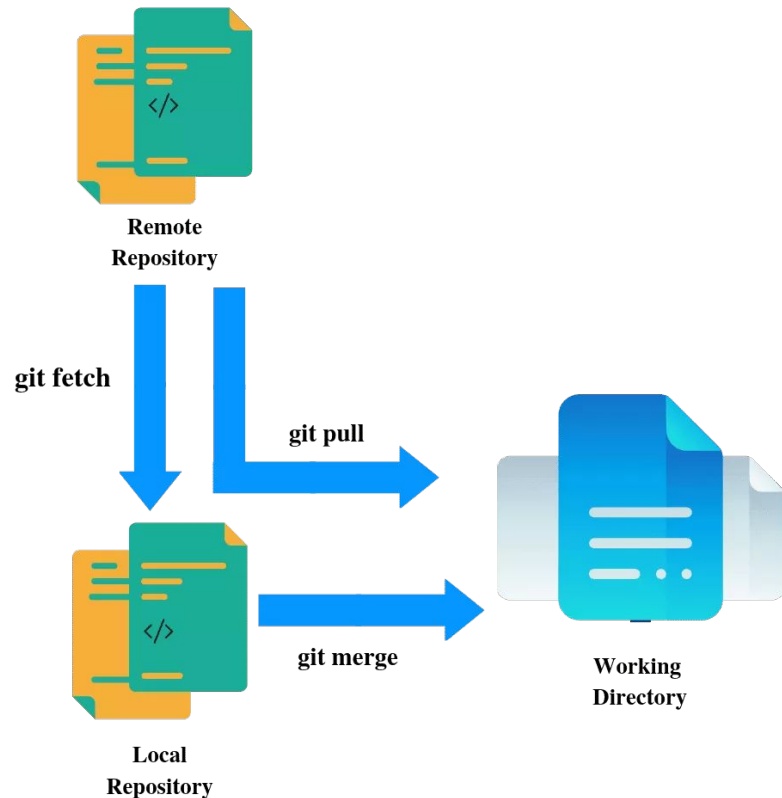


remotes

- **git fetch** → aggiorna lo stato del remote
- **git pull** → aggiorna lo stato del remote e unisce le modifiche.
equivalente a fetch + merge
- **git push** → notifica il remote del branch e commit corrente



Tutti i comandi possono specificare il remote a cui si riferiscono: il default è **origin**.



git pull

Se le storie del branch remoto e del branch locale fossero divergenti, git pull avrebbe tentato di riconciliarle.

La strategie possono essere specificate nel comando stesso, o può essere impostato un default configurando l'impostazione `pull.rebase`. Le più diffuse:

- `--rebase`: analogo a `git rebase origin/nome-branch`
- `--no-rebase`: utilizza merge. Analogo a `git merge origin/nome-branch`
- `--ff-only`: annulla l'operazione se non è possibile un fast-forward

N.B: git pull esegue già git fetch, non è necessario eseguire git fetch manualmente



riscrivere la storia

riscrivere la storia

Gli oggetti e la storia di git sono immutabili e validati da hash. Riscrivere la storia non significa modificare la storia esistente, ma **rimpiazzarla con storia nuova**.

Riscrivere la storia può essere utile per:

- mantenere una **storia più pulita e lineare**
- **risolvere conflitti**
- **rimuovere dati sensibili** aggiunti per sbaglio

Ma riscrivere la storia significa anche perdere la storia precedente: non va fatto con leggerezza!



In caso di errori, il reflog permette di recuperare lo stato precedente.

riscrivere la storia

I 3 comandi più comuni legati alla riscrittura della storia sono:

- `git commit --amend` → modifica l'ultimo commit
- `git rebase` → modifica la storia del branch
- `git push --force` → sovrascrive la storia del branch remoto con la propria

Bisogna prestare particolare attenzione all'uso di questi comandi quando **la storia che si rimpiazza è già stata condivisa**: si rischia di generare conflitti non sempre facili da risolvere o perdere modifiche e dati importanti.

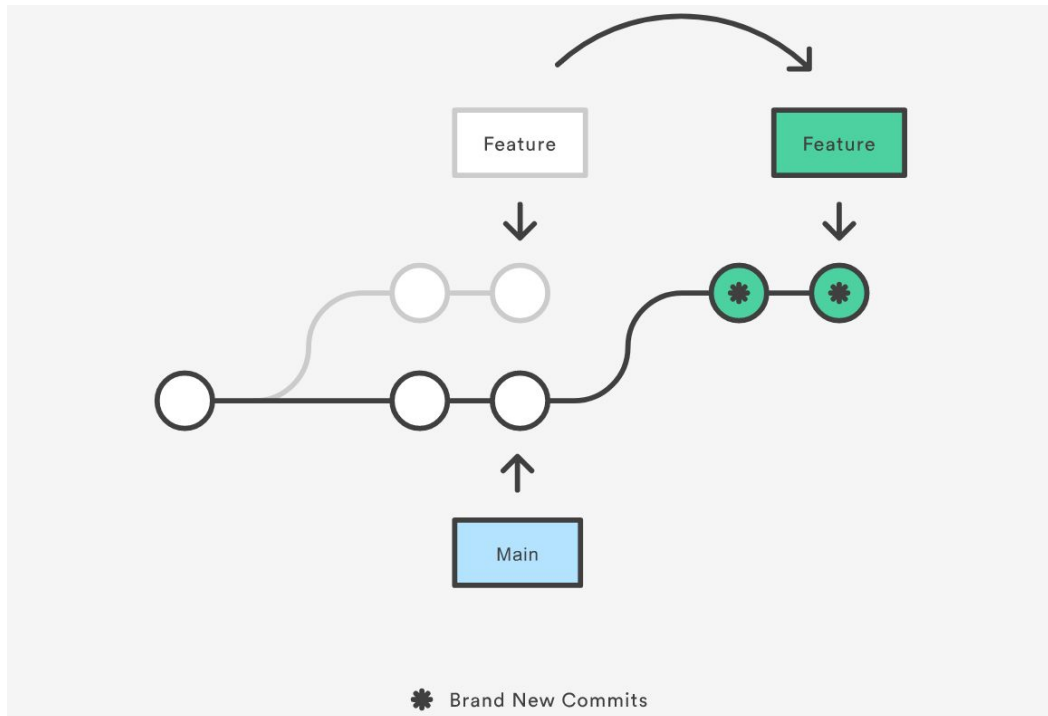


git rebase

Il comando **git rebase <target>** permette di “rifondare” il proprio branch a partire da un altro branch o commit.

Le modifiche contenute nei commit unici al proprio branch vengono applicate di nuovo, in coda a quelle del target.

Il processo crea nuovi commit con nuovi hash, anche se contenenti le stesse modifiche.



git rebase

git rebase viene spesso usato per portare su un branch di lavoro **feature** modifiche apportate sul branch di riferimento **main** mentre si lavorava su **feature**, senza la creazione di merge commit inutili.

L'effetto finale sarà come se **feature** fosse stato creato da **main** adesso e non prima che le nuove modifiche esistessero su **main**.

A questo punto, fare il merge di **feature** su **main** risulterà in un **fast-forward** senza merge commit, mantenendo la **storia lineare**.

Inoltre, a volte applicazioni e servizi di hosting git che permettono di eseguire merge possono richiedere all'utente di fare un rebase in caso di conflitti che non sono in grado di risolvere automaticamente.



git push --force

git push è efficace soltanto se le storie del branch locale e remoto non sono divergenti (cioè se il push è in fast-forward).

Nel caso la parte di storia già condivisa con il remote sia stata riscritta (da **git rebase**, **git commit --amend** o altri comandi), bisognerà forzare l'operazione con: **git push --force**.

L'operazione **sovrascrive forzatamente** la storia sul remote con la propria.

 *Attenzione! Possibile perdita di dati!* 

Si rischia di sovrascrivere il lavoro di altri, se si esegue su un branch condiviso.



git push --force

⚠ *Attenzione! Possibile perdita di dati!* ⚠

È fortemente sconsigliato l'uso su branch **su cui lavorano attivamente anche altre persone**, pena generazione di conflitti difficili da risolvere.



approfondimenti

approfondimenti

- [Git for Ages 4 and Up](#) - Eccezionale presentazione che illustra, con analoghi visivi, le principali operazioni alla base del funzionamento di git.
- [Git Immersion](#) - Tutorial guidato
- [Git Internals v2](#) - Free Ebook sul funzionamento, anche a basso livello, di git.
- Il [subreddit di git](#) ([Accedi tramite libreddit](#)) - Stimoli interessanti nei thread, e tanti materiali nella sidebar



need help?

dove trovo aiuto?

Il web è tuo amico!

La documentazione è tua amica!

È possibile scaricare gratuitamente l'ebook Pro Git, la bibbia di Git
<https://git-scm.com/book/en/v2>

