

Git 1 comandi base

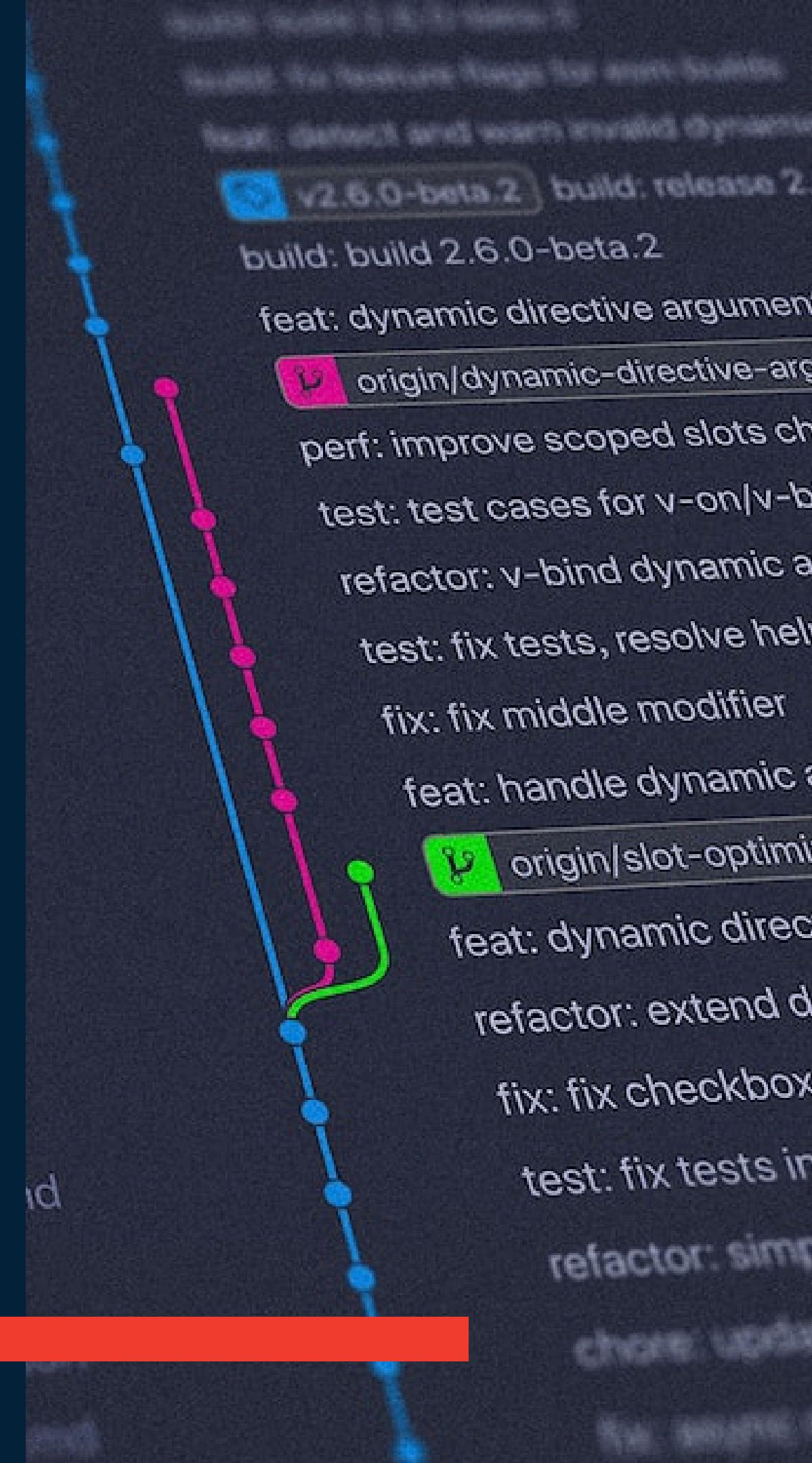
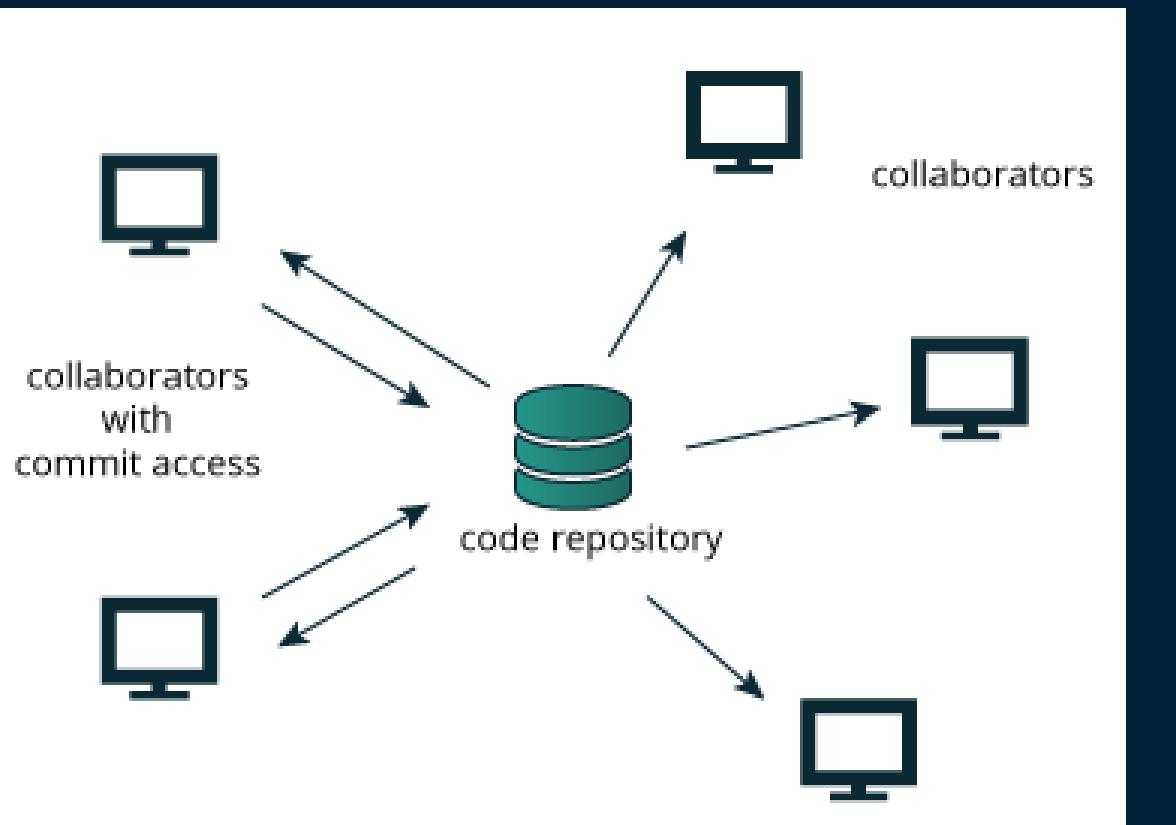
risorse.vercel.app/lab/2025

Alice Benatti, Mattia Graziani



Cos'è git?

- **sistema di versionamento** = tenere traccia delle versioni dei file, ottenendo una cronologia di tutte le modifiche
- **distribuito** = chiunque ha una copia completamente locale della *repository*



Perché usarlo?

- Cronologia trasparente su tutti i cambiamenti dei file
- Ripristinare subito un qualsiasi **stato precedente**
- Collaborare a copie dinamiche diverse dello stesso progetto

"FINAL".doc



FINAL.doc!



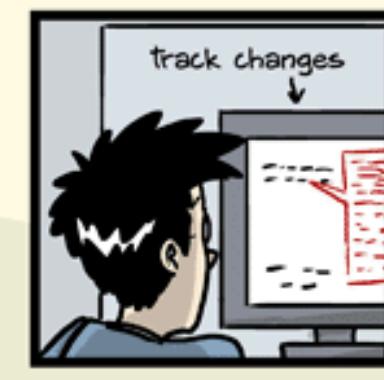
FINAL_rev.2.doc



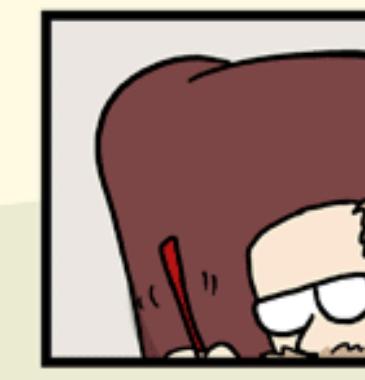
FINAL_rev.6.COMMENTS.doc



FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments7.
corrections9.MORE.30.doc



FINAL_rev.22.comments49.
corrections.10.#@\$%WHYDID
ICOMETOGRAD SCHOOL?????.doc

JORGE CHAM © 2012

Installare git Windows

- via Command Line con *winget*
`winget install --id Git.Git -e --source winget`
- scaricare dal [link del sito ufficiale](#)

[Click here to download](#) the latest (2.42.0) 64-bit version of Git for Windows. This is the most recent [maintained build](#). It was released [about 2 months ago](#), on 2023-08-30.

[Other Git for Windows downloads](#)



Installare git Linux

con Ubuntu/Mint/Debian

```
$ sudo apt-get update  
$ sudo apt-get install git
```

con Fedora

```
$ yum install git (up to Fedora 21)  
$ dnf install git (Fedora 22 and later)
```

[[per altre distribuzioni clicca qui](#)]



Installare git MacOS

Ci sono diversi package manager che permettono di installare facilmente git, scegli quello che preferisci:

- ***Homebrew***

Installa [homebrew](#) se non lo hai già, e lancia il comando:

```
$ brew install git
```

- ***MacPorts***

Installa [MacPorts](#) se non lo hai già, e lancia il comando:

```
$ sudo port install git
```

- ***Xcode***

Apple fornisce un pacchetto binario Git con [Xcode](#).

[per ulteriori info seguite la [guida sul sito ufficiale](#)]



Configurazione Profilo

Configurare *git* con i vostri dati:
usate la mail che preferite, sarà poi associata con il vostro user.name alle varie operazioni
che svolgerai con git.

```
$ git config --global user.name "Nome Cognome"  
$ git config --global user.email "nome.cognome@studio.unibo.it"  
$ git config --global init.defaultBranch main
```

repository



repository è un progetto gestito con git

È come una **cartella** in cui contenere tutti i file di un progetto.

Se ne possono creare tante quante volete.

git init



00

Per inizializzare una *repository* Git vuota in una nuova <directory>;
all'interno di una *directory* lanciamo:

```
$ git init [<directory>]
```

Nota

1. I file interni di git sono nella sottocartella <directory>/.git/
2. Se <directory> non è specificata, viene usata la cartella corrente.

Per convenzione nelle slide useremo comandi shell (\$): quando ci saranno [qualcosa], intenderemo che è facoltativo; quando useremo < qualcosa >, intenderemo che è una parte da completare



.git

cartella autogenerata al momento della creazione di una repository con git

È dedicata a tutti i file di git, non deve essere modificata!!

Serve per tenere traccia di:

- commit
- branch
- files
- modifiche
- tag
- worktree

Creiamo un file

01

```
$ nano README.md
```

Scriviamo dentro al file:
“Il nostro primo file”

Come salviamo questo stato della repository?

Vogliamo poter tornare allo stato attuale in un qualsiasi momento futuro

Creiamo un file

01

```
$ nano README.md
```

Scriviamo dentro al file:
“Il nostro primo file”

Questo file viene considerato da git come untracked
ovvero non è mai stato tracciato da git

02

git add

Aggiunge le modifiche specificate in <path> agli *staged files*

```
$ git add [<path>...]
```

Per aggiungere il nostro file eseguiamo:

```
$ git add README.md
```

git commit

03

Registra le modifiche agli *staged files* commentandole con un <msg>

```
$ git commit [-m <msg>]
```

Se non uso *-m*, viene aperto un *editor* di testo in cui posso specificare il messaggio,
!! se l'*editor* di testo non è specificato lanciate il comando:

```
$ export EDITOR=nano
```



Ricominciamo

Creiamo un altro file

00

```
$ nano HelloWorld.cpp
```

```
#include <iostream>

int main() {
    std::cout << "Hello World\n";
}
```

git status

01

Mostriamo lo stato attuale della nostra *repository*

\$ git status

Dovreste vedere una situazione del genere:

```
$ git status  
[...]  
Untracked files:  
  HelloWorld.cpp
```

untracked files = tutti i file di cui al momento git non sta tenendo conto

git add

02

Aggiunge i file scelti agli *staged files*

```
$ git add HelloWorld.cpp
```

Dovreste vedere una situazione del genere:

```
$ git add  
[...]  
Changed to be committed:  
  new file: HelloWorld.cpp
```

Abbiamo aggiunto *HelloWorld.cpp* alla **Staging Area**

A questo punto git è a conoscenza del nostro file e monitorerà tutte le sue modifiche future.

Modifichiamo il file

03

```
$ nano README.md
```



Il nostro primo file
Progetto HelloWord in C++

git status

04

Mostriamo lo stato attuale della nostra *repository*

\$ git status

Dovreste vedere una situazione del genere:

```
$ git status  
[...]  
Changes not staged for commit:  
modified: README.md
```

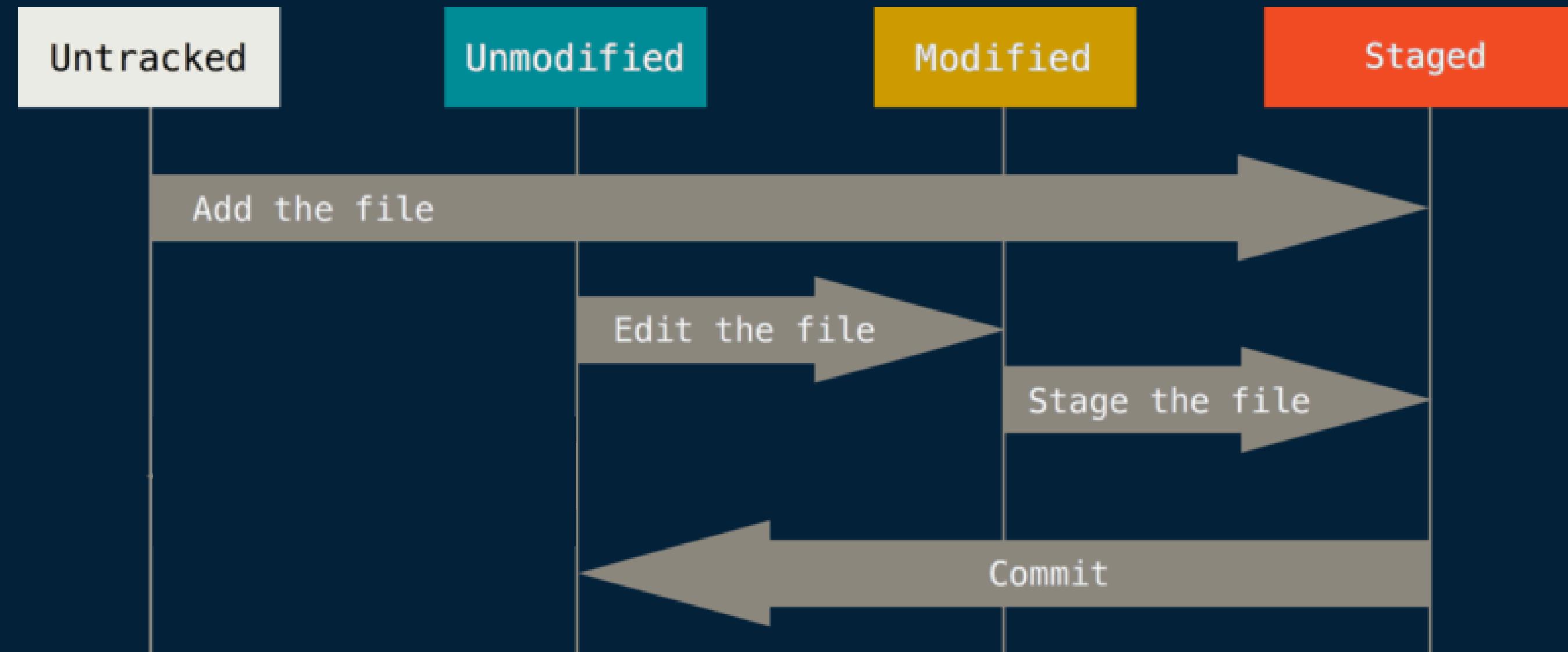
Modified Area = tutti i file che sono stati modificati rispetto al del commit precedente

Stato dei file

I file di una repository sono sempre in esattamente uno dei seguenti quattro stati:

1. ***untracked*** non tracciato da git
2. ***unmodified*** non modificato rispetto all'ultima “istantanea” di git
3. ***modified*** modificato rispetto all'ultima “instantanea” di git
4. ***staged*** modificato rispetto all'ultima “instantanea” di git e pronto ad essere registrato

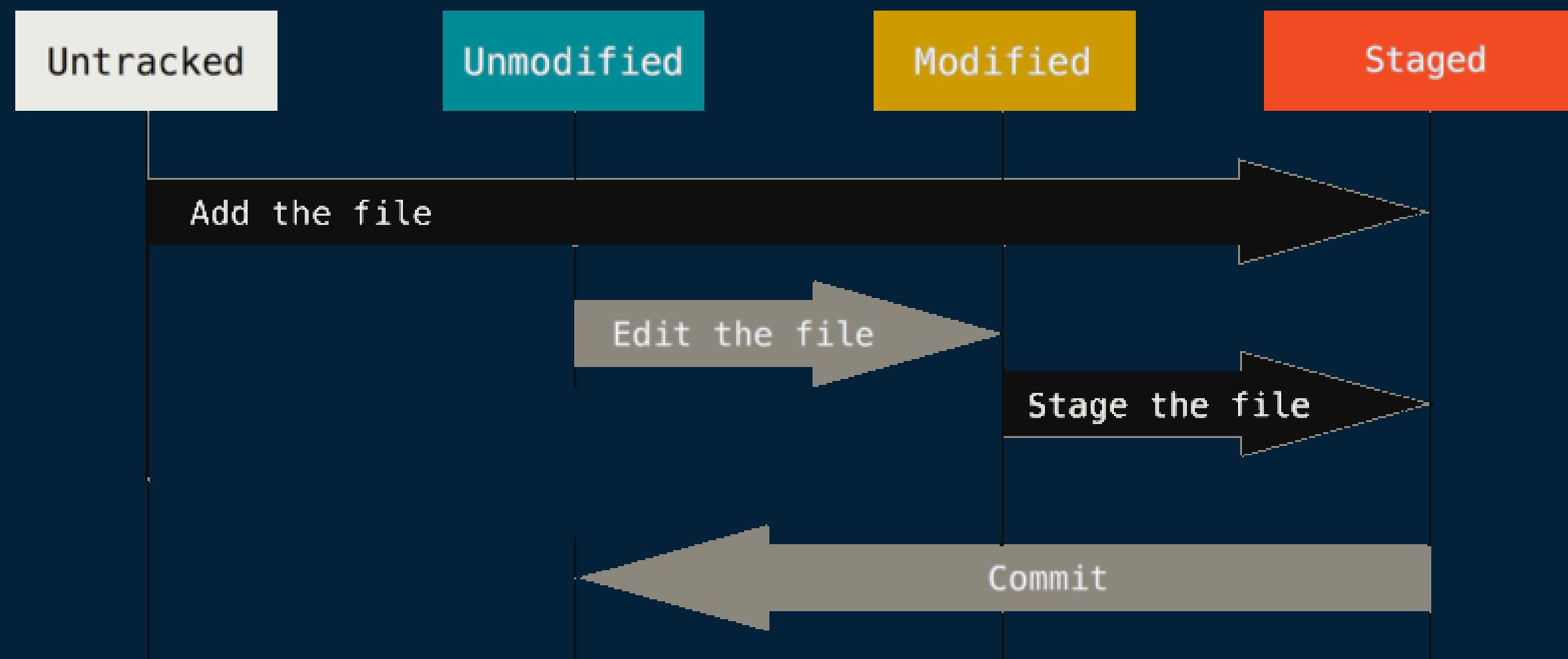
git status



git add

Aggiunge <path_specifico> (non tracciato o modificato) agli *staged files*:

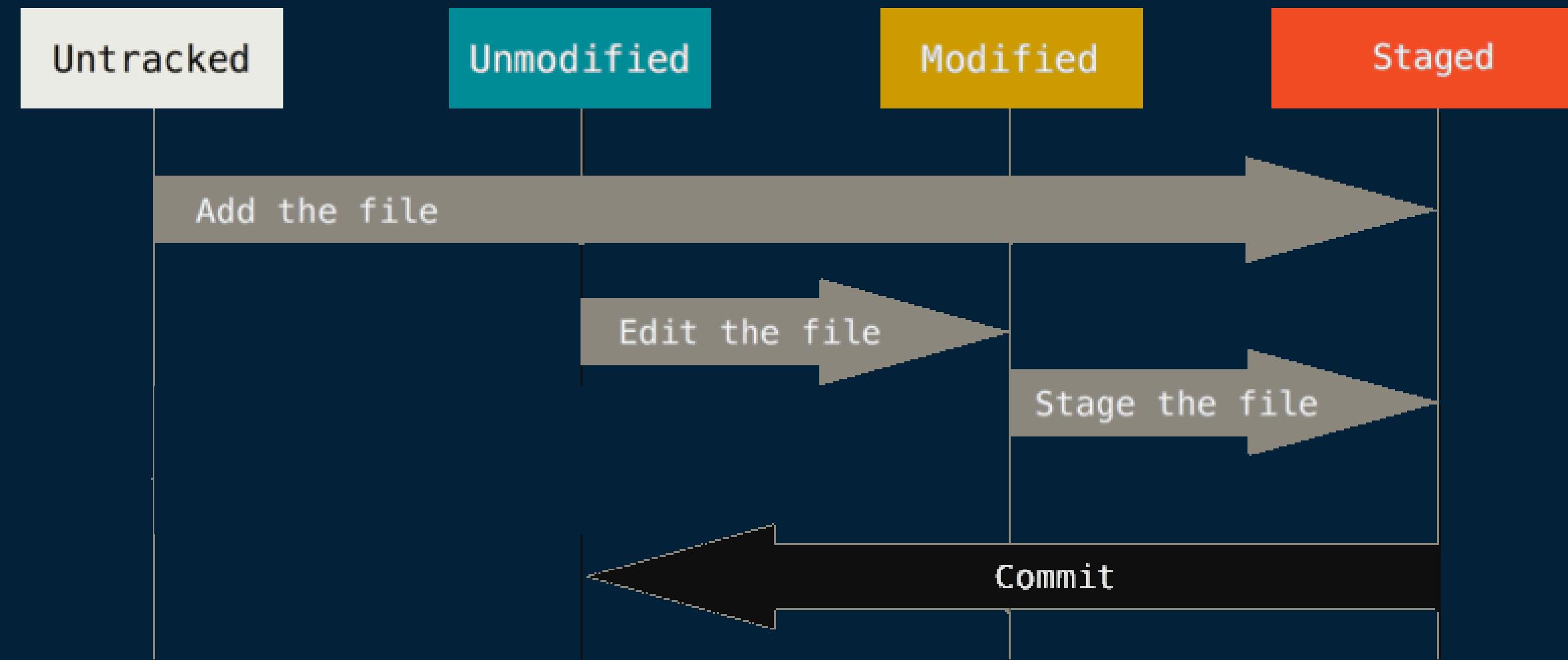
```
$ git add [<paths>...]
```



git commit

Registra le modifiche agli *staged files* commentandole con un *<msg>*

```
$ git commit [-m <msg>]
```



git commit

buone abitudini

1. fare commit quando si è alla **fine di una determinata modifica**:

il mio obiettivo: aggiustare un bug

commit quando dopo varie modifiche al progetto riesco a raggiungere il mio obiettivo

[molto probabilmente non sarà l'ultimo commit per il nostro obiettivo, ma dobbiamo cercare di farne il meno possibile]

2. **Convenzione**: dare nomi riconoscibili ai commenti del commit

- add
- feat
- fix
- docs
- style
- [...]



breve descrizione

05

Aggiungiamo tutte le modifiche alla staging area con:

```
$ git add .
```

Nota

Con “.” aggiunge tutti i file presenti nella directory che sono stati modificati o che sono nell'**untracked area**.

Committiamo:

```
$ git commit -m "add: helloworld.cpp"
```

Verifichiamo lo status:

```
$ git status
```

Dovreste vedere una situazione del genere:

```
$ git status
Nothing to commit, working tree clean.
```

Unmodified Area = avendo committato tutti i file, non abbiamo attualmente altre modifiche sui file.

Creiamo una cartella all'interno della repository:

```
$ mkdir utils
```

Creiamo un file all'interno della sotto cartella appena creata:

```
$ touch utils/docs.md
```

e creiamone uno nella cartella principale:

```
$ touch config.md
```

Verifichiamo lo status:

```
$ git status
```

Dovreste vedere una situazione del genere:

```
$ git status
[...]
Untracked files:
  config.md
  utils/
```

06

Committiamo tutte le modifiche:

```
$ git add .
```

```
$ git commit -m "feat: utils & config"
```

git log

Mostriamo il registro dei commit:

```
$ git log [--graph]
```

Notiamo che sono visibili tutti i commit ordinati per data.

Ogni commit ha un numero identificativo **hash code**, un autore, la data e testo.

```
commit f81e2661ec68130d6627277f47d3b3f73b2c9f0d (HEAD -> main)
Author: John Doe <johndoe@email.com>
Date:   Tue Mar 15 10:00:00 2023 -0400

    Add new feature

commit 8105e5b6fb5f6b166c6de5c9d4d4db4b44f71aa7
Author: Jane Doe <janedoe@email.com>
Date:   Mon Mar 14 10:00:00 2023 -0400

    Fix bug in existing feature

commit b13f6098dd1cdaec24e3c3c3e9d62e7bb56b38ec
Author: John Doe <johndoe@email.com>
Date:   Sun Mar 13 10:00:00 2023 -0400

    Initial commit
```

hash code ↗

Nota

1. usando *--graph*, il registro è rappresentato come un grafo

Modifichiamo il file

0'7

```
$ nano HelloWorld.cpp
```

```
#include <iostream>

int main() {
    std::cout << "Hello World\n";
    return 0;
}
```

git diff

08

Mostra le differenze presenti nei file rispetto a quelli di git committati precedentemente

\$ git diff

Dovreste vedere una situazione del genere:

```
$ git diff
diff --git a/helloWorld.cpp b/helloWorld.cpp
index df92bc7..8b66388 100644
--- a/helloWorld.cpp
+++ b/helloWorld.cpp
@@ -2,4 +2,5 @@

```

```
int main() {
    std::cout << "Hello World\n";
+    return 0;
}
```

git diff

08

Mostra le differenze presenti nei file rispetto a quelli di git committati precedentemente

```
$ git diff
```

Nota

Git diff mostra solo le differenze dei file **non ancora** nella Staging Area,

per mostrare le differenze dei file **nella Staging Area** usiamo: `$ git diff --cached`

Si può specificare su quali file mostrare le differenze aggiungendo i path in fondo al comando:

```
$ git diff [--cached] <path_del_file>
```

```
$ git diff [--cached] <path_del_file1> <path_del_file2>
```

08

Nel nostro progetto, noi aggiungiamo le modifiche fatte alla staging area dato che c'era una dimenticanza!

```
$ git add HelloWorld.cpp
```

Proviamo ora a fare:

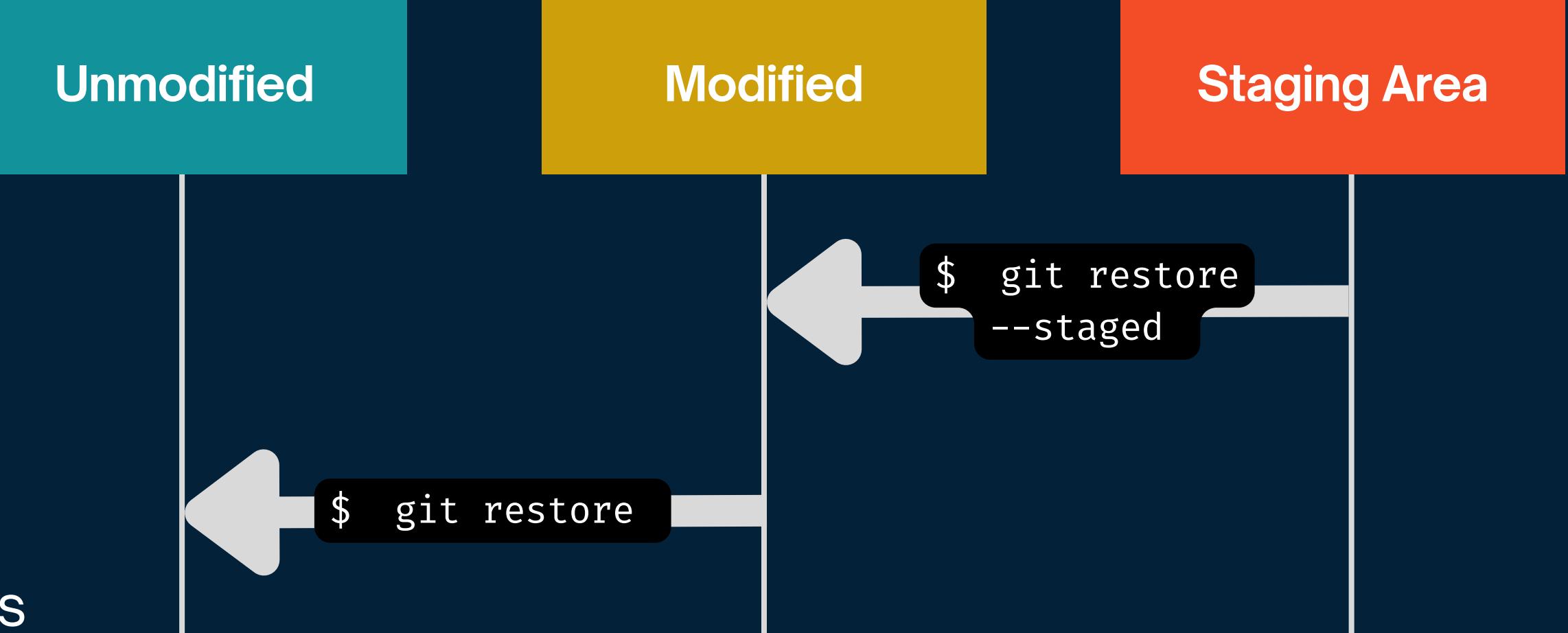
```
$ git diff --cached
```

e committiamo: `$ git commit -m "fix: helloworld.cpp"`

git restore

Ripristina un file alla versione dell'ultimo commit

```
$ git restore [--staged] <path_file>
```



Attenzione!

facendo `$ git restore`

le modifiche effettuate sui files
verranno perse

git restore

Ripristina un file ad una specifica versione/commit

Quindi per togliere *HelloWorld.cpp* dalla staging area faremo:

```
$ git restore --staged HelloWorld.cpp
```

Se volessimo ripristinare il file all'ultima versione registrata su git useremo:

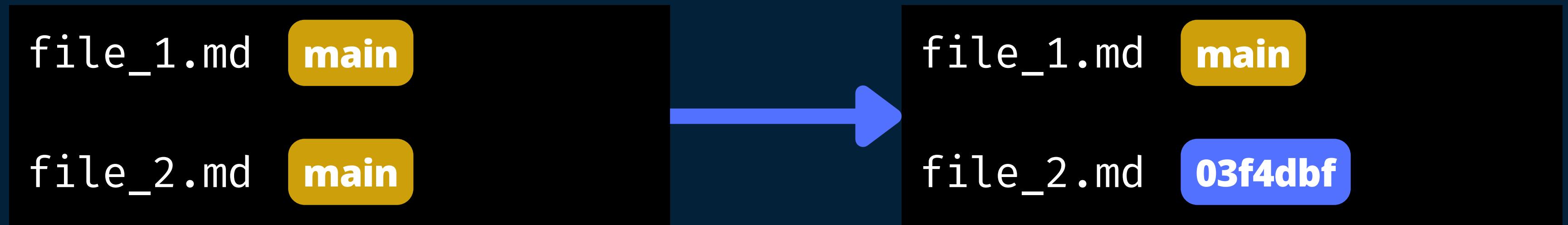
```
$ git restore HelloWorld.cpp
```

git restore

Ripristina un file ad una specifica versione/commit

```
$ git restore --source=<commit/branch> <path_file>
```

Se il file è modificato attenzione che verrà sovrascritto



```
$ git restore --source=03f4dbf file_2.md
```

git checkout

Permette di spostarsi su un altro commit, ma non solo...

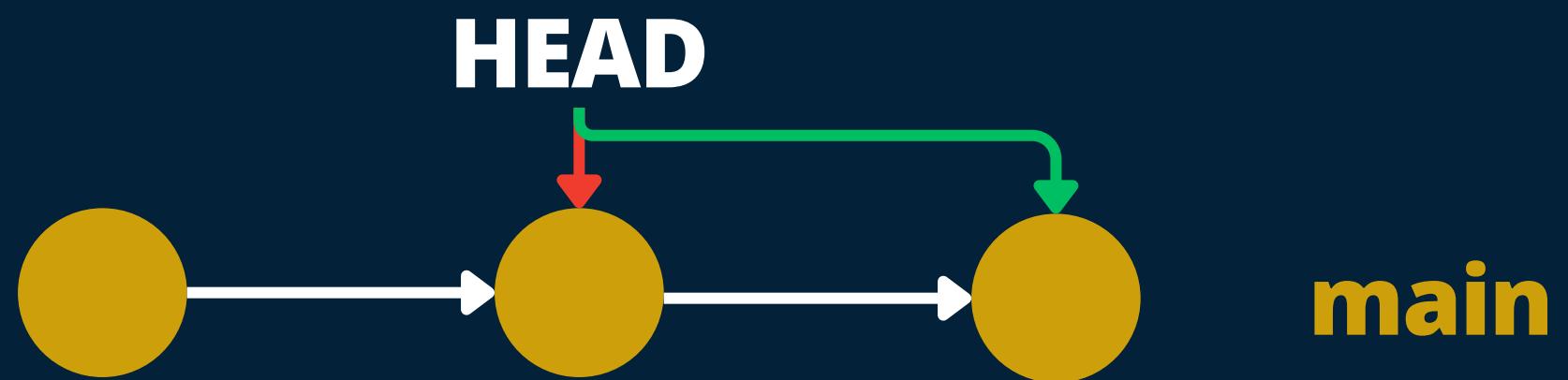
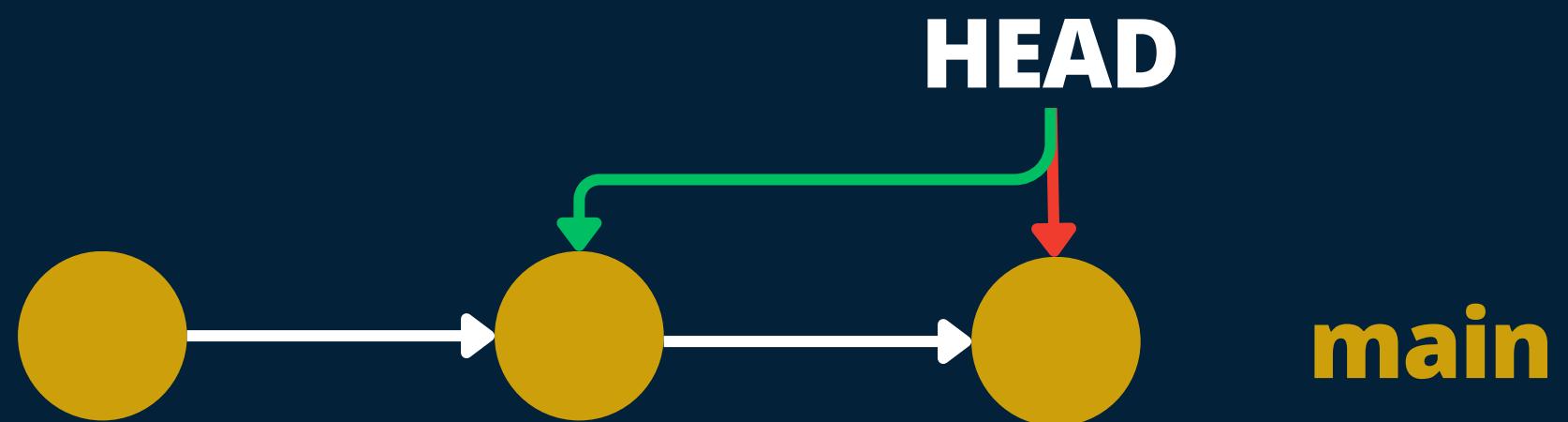
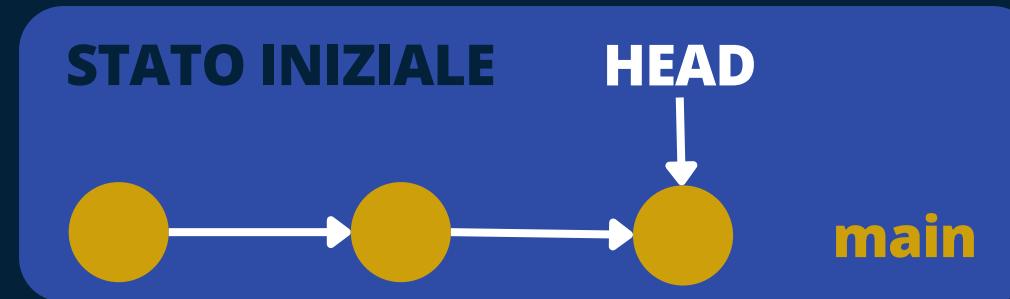
```
$ git checkout <hash_code>
```

HEAD punta sempre al commit corrente

Non si può fare se ci sono file modificati all'interno della directory...

Torniamo indietro nel commit più recente:

```
$ git checkout main
```



git stash

Mette da parte la staging area che non vogliamo committare ora

```
$ git stash
```

Per riprendere le nostre modifiche usiamo:

```
$ git stash pop
```

Attenzione

Quando si fa *checkout* è necessario che nella directory corrente non ci siano modifiche, per evitare che vengano sovrascritte.

Così le salviamo in un “commit temporaneo” che è semplice da riprendere.

Mani in pasta

00

Creiamo una nuova repository

```
$ git init AppStaff
```

Creiamo alcuni commit per avere una storia (history) su cui sperimentare:

```
$ touch README.md
```

```
$ git add .
```

```
$ git commit -m 'add: README.md'
```

Facciamo la stessa cosa creando un nuovo file import.md e committiamo.

01

Modifichiamo **README.md** aggiungiamo del contenuto come:

```
$ nano README.md
```

```
 Lorem ipsum dolor sit amet, consectetur adipiscing  
elit. Nulla eget accumsan tortor. Nam lacinia  
lorem vitae est tristique ullamcorper. Suspendisse  
potenti. Suspendisse potenti. Aenean eget risus eu  
turpis ultricies porta fermentum vitae urna.  
Vivamus sit amet aliquet massa.
```

*Vogliamo salvare questo stato della repository nella history di git,
come facciamo?*

Purtroppo per errore, mentre ci passa il gatto sulla tastiera ci cancella dei caratteri e riesce anche a committarli:

```
$ nano README.md
```

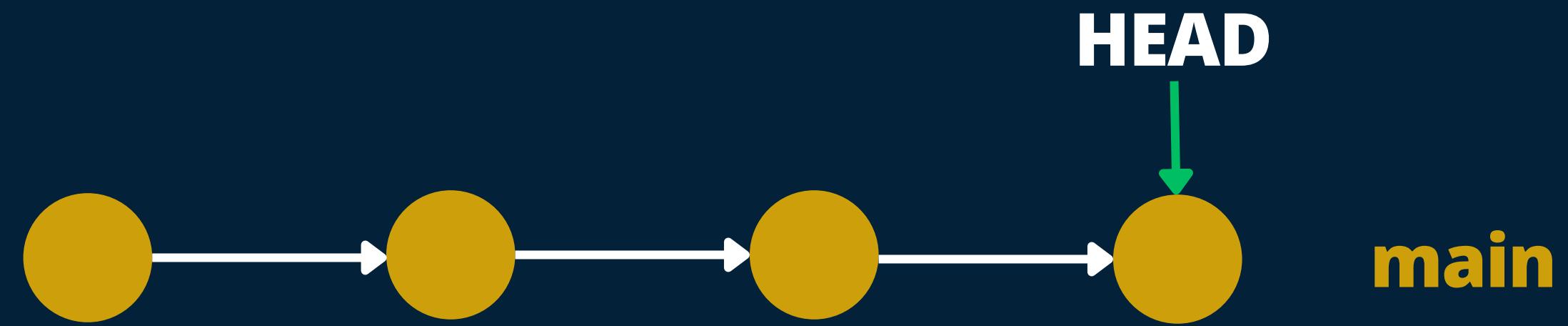
01

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla **eget** accumsan tortor. Nam lacinia lorem vitae est **tristique** ullamcorper. Suspendisse **potenti**. Suspendisse potenti. Aenean eget risus eu turpis ultricies porta fermentum vitae urna. Vivamus sit amet aliquet massa.

03

A questo punto abbiamo fatto 4 commit!

Facendo *git log* dovremmo vedere una situazione di questo tipo:



Facciamo una modifica: scriviamo qualcosa dentro al file vuoto

```
$ nano import.md
```

```
# This is a file for imports
```

Se in realtà non volessimo queste modifiche?

In progetti reali, può capitare di implementare qualcosa ma che ci accorgiamo contenere troppi bug o non sono semplicemente modifiche che vogliamo più.

Per rimuoverli possiamo:

- andare manualmente a toglierli file per file, rischiando di perdere dei pezzi
- **ripristiniamo lo stato della repository all'ultimo commit:**

```
$ git restore <path_da_ripristinare>
```

04

Aggiungiamo del testo nuovo a README.md

```
$ nano README.md
```

```
# AppStaff - our first app with ADMStaff
```

```
This is a new project with ADMStaff Team.
```

***Non ci ricordiamo però com'era il file prima di queste modifiche!
Come si può fare?***

04

***Non ci ricordiamo però com'era il file prima di queste modifiche!
Come si può fare?***

Abbiamo diverse opzioni

```
$ git diff
```

```
$ git log <file>  
$ git checkout <hash_code>
```

```
$ git log -p <file>
```

...

04

Non funziona!

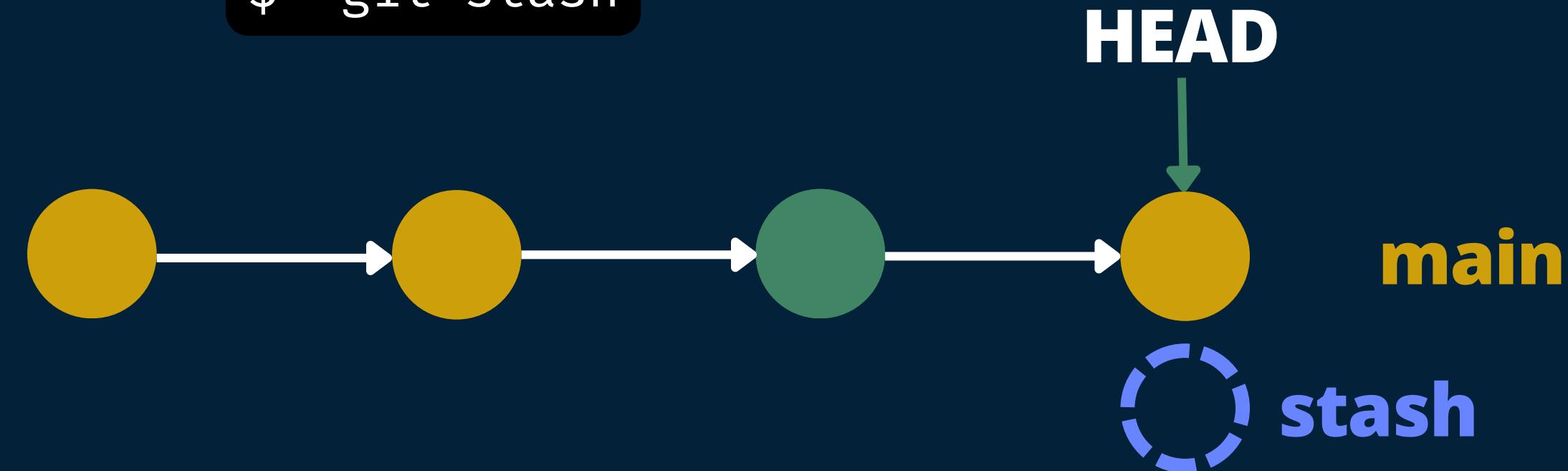
Non possiamo tornare indietro perché c'è un conflitto sul file install.yaml!

A questo punto possiamo scegliere di:

- committare lo stato attuale - il progetto non è in uno stato funzionante
- **mettere da parte le modifiche e lavorarci dopo**

Facciamo quindi:

```
$ git stash
```



05

Viaggiare “indietro nel tempo”

Una volta trovato l’hash del commit in cui saltare usiamo:

```
$ git checkout <hash>
```

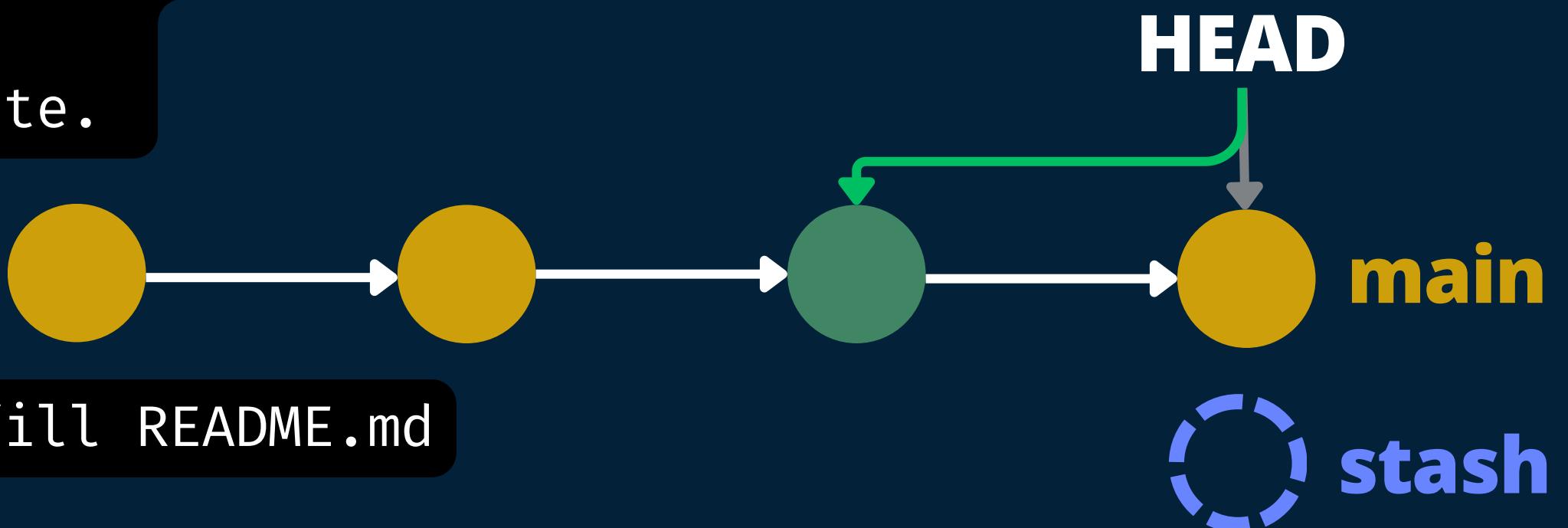
A questo punto siamo in una situazione di questo tipo:

Note: switching to 'be3a85[...]aca678962'.

You are in 'detached HEAD' state.

[...]

HEAD is now at be3a85 chore: fill README.md



Viaggiare “indietro nel tempo”

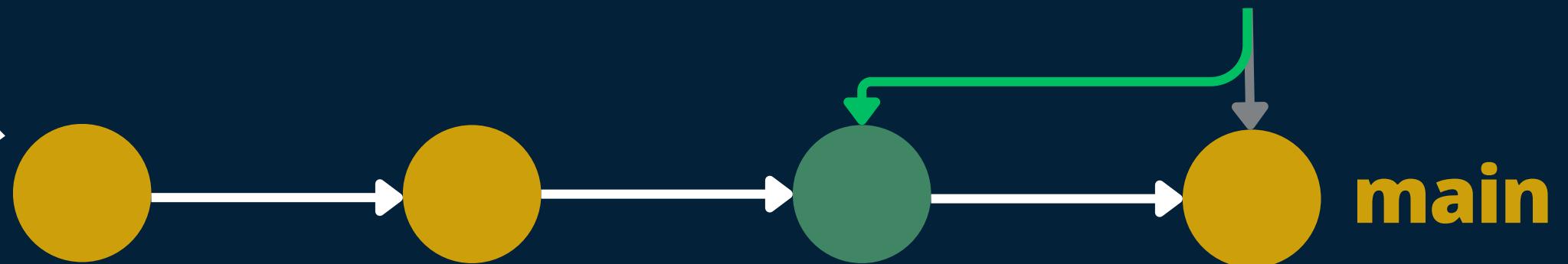
Quando siamo in un qualsiasi stato della repository, abbiamo accesso a tutto:

- visualizzare file
- modificare file
- creare file
- eliminare file

Di solito però vale sempre il **principio di causalità** (o effetto farfalla):

*«Qualunque cosa tu faccia,
non cambiare nulla del passato.»*

HEAD



Con questa filosofia, visitiamo il file README.md, e vediamo il suo stato iniziale

```
$ cat README.md
```

Viaggiare “indietro nel tempo”

Vediamo ora di ottenere una versione di README.md con:

- modifiche che abbiamo messo da parte nello stash
- lo stato del file nel commit che abbiamo visitato

Ritorniamo all’ultimo commit in cui eravamo, quello puntato da main

```
$ git checkout main
```

Ripristiniamo lo stato del file README.md con quello del primo commit:

```
$ git restore --source=<hash_commit> install.yaml
```

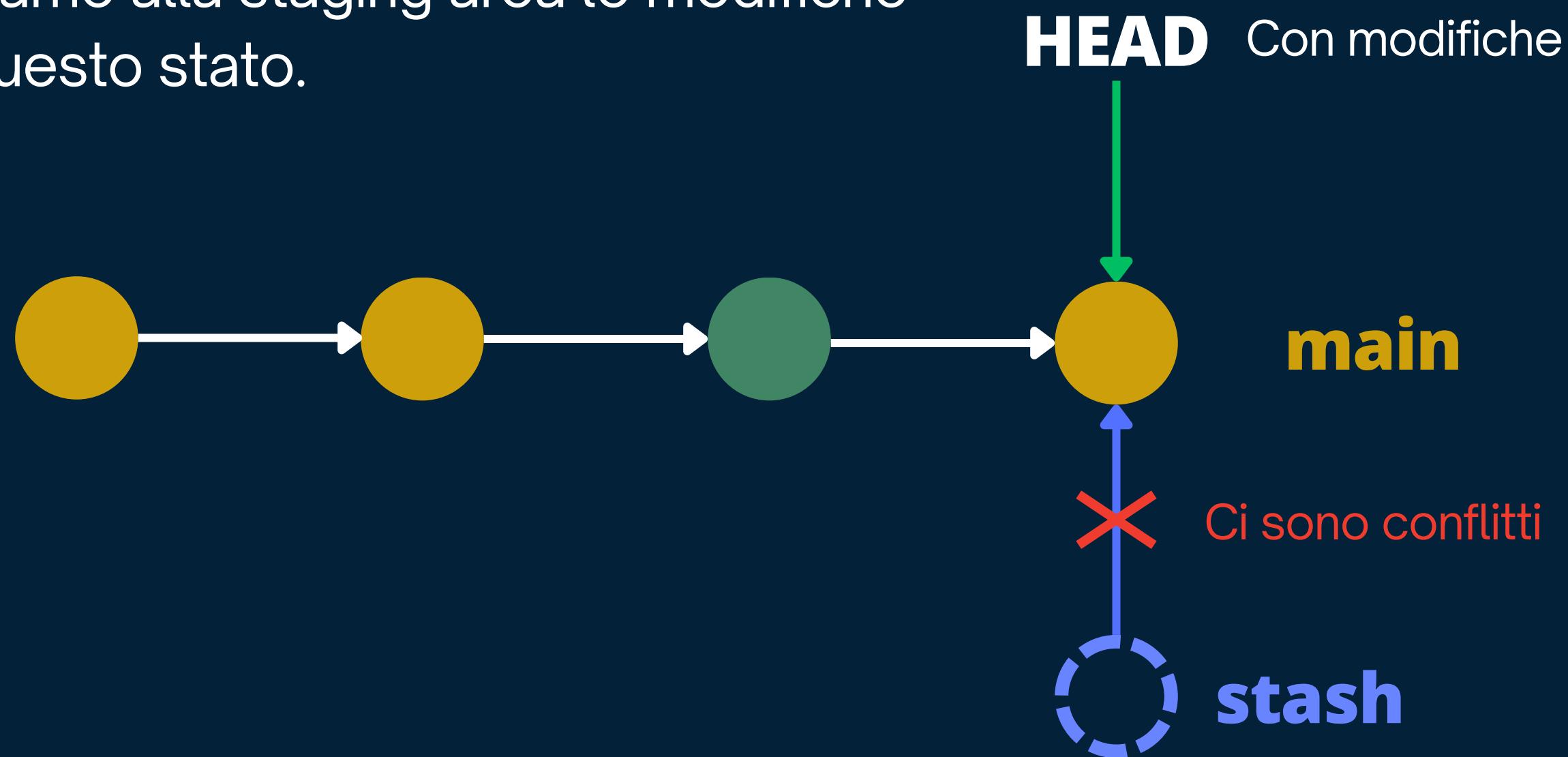
Proviamo ora a riprenderci le modifiche che avevamo lasciato da parte prima per creare una soluzione finale complessiva:

```
$ git stash pop
```

NON possiamo farlo!!!

Abbiamo bisogno di **committare** i cambiamenti nello stato attuale per poter prendere i file dallo stash.

Quindi aggiungiamo alla staging area le modifiche e registriamo questo stato.



A questo punto possiamo riprenderci le modifiche:

\$ git stash pop

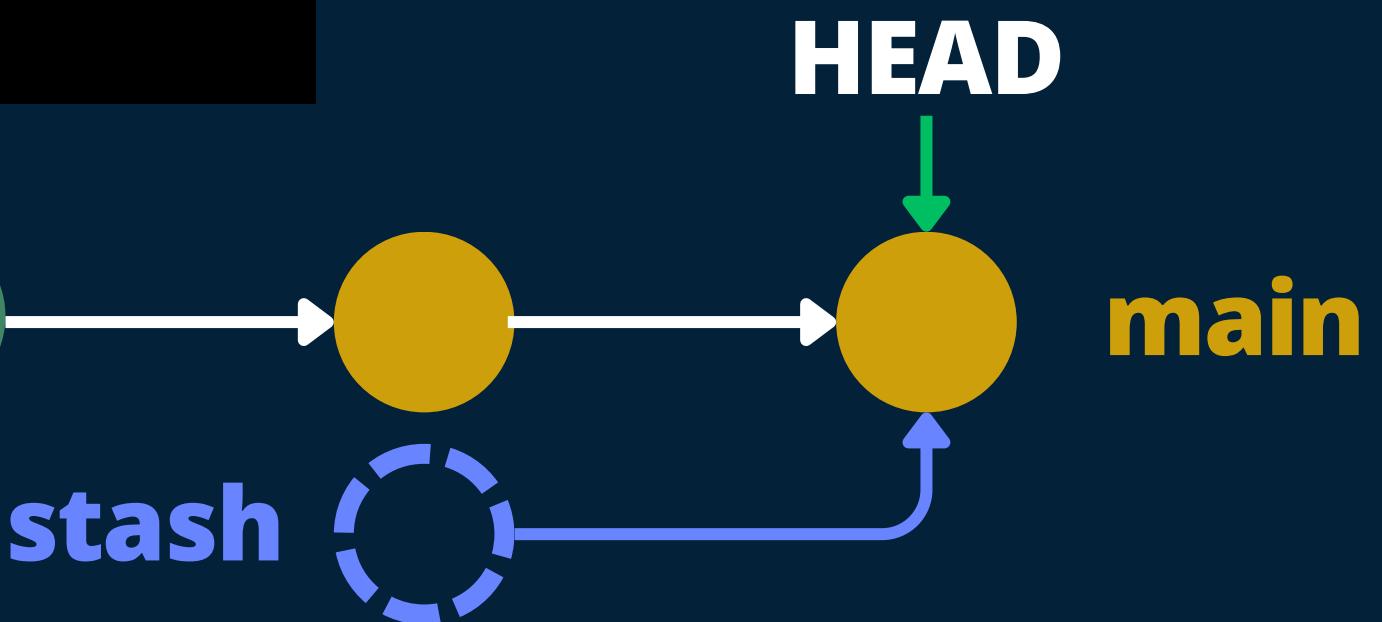
o

\$ git stash apply
\$ git stash drop

Notiamo però che ci esce questo output:

```
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
[...]
Unmerged paths:
[...]
both modified: README.md
```

The stash entry is kept in case you need it again.



Ci dice che abbiamo un **conflitto!** Niente panico, possiamo risolverlo!
Git ci dice che README.md ha due versioni differenti.
Aprendo il file dobbiamo modificarlo con quello che vogliamo salvare.

08

```
>>>>> HEAD  
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla eget  
accumsan tortor. Nam lacinia lorem vitae est tristique ullamcorper.  
Suspendisse potenti. Suspendisse potenti. Aenean eget risus eu turpis  
ultricies porta fermentum vitae urna. Vivamus sit amet aliquet massa.  
=====
```

```
# AppStaff - our first app with ADMStaff
```

```
This is a new project with ADMStaff Team.  
<<<<<< stash
```

Nota Le righe aggiunte da git che iniziano con <<<<<, =====, >>>>> devono essere tolte.
Approfondiremo bene i conflitti nel prossimo laboratorio.

09

Ora abbiamo il nostro progetto funzionante con le modifiche che volevamo.
Possiamo salvare questo stato con un commit, dato che il conflitto è stato risolto.

Flag speciali

Scopri tu cosa fanno di magico ✨

```
$ git add -p [<path>]
```

```
$ git add -u
```

```
$ git commit -am <msg_commit>
```

```
$ git commit --ammend
```

```
$ git log --grep='<string>'
```

```
$ git stash -u
```

>ADM
staff

LABORATORI FRA PARI

AA 2025-2026

GIT 2

CONDIVISIONE
REMOTA

20/11 16:00-18:00

AULA E2

via mura Anteo Zamboni 2B

GITHUB 1

ISSUE
PULL REQUEST
CONFLITTI

27/11 16:00-18:00

AULA E2

via mura Anteo Zamboni 2B

GITHUB 2

CI/CD
ACTIONS
GIT LFS

04/12 16:00-18:00

AULA E2

via mura Anteo Zamboni 2B



PORTA LA TUA
CURIOSITÀ'

