git

introduzione

DIGITAL DREAMERS
ASSOCIAZIONE

cos'è git

GIT - the stupid content tracker

The name "git" was given by Linus Torvalds when he wrote the very first version. He described the tool as "the stupid content tracker" and the name as (depending on your mood):

- **random three-letter combination** that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronounciation of "get" may or may not be relevant.
- **stupid**. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- **"global information tracker"**: you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- **"goddamn idiotic truckload of sh*t"**: when it breaks

https://github.com/git/git

cos'è git

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

- tiene traccia dello storico dei cambiamenti fatti al progetto
- mostra le differenze tra vari stati del progetto
- divide lo sviluppo del progetto in più “rami” indipendenti, branches
- permette di ricombinare i branches tramite 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 GITHUB/GITLAB

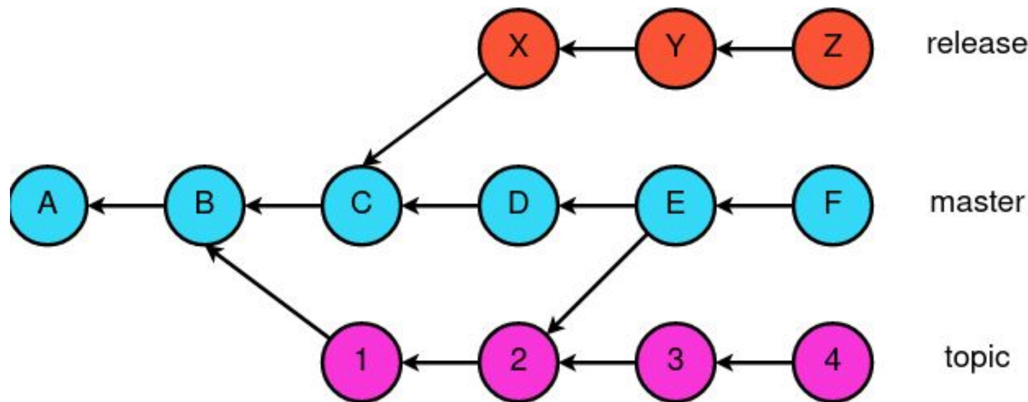
Git è lo strumento
Github/GitLab sono servizi che ospitano progetti gestiti con git

Forniscono un repository nel cloud con cui sincronizzare i tuoi progetti git, insieme ad altri servizi tra cui CI/CD, wiki, ticketing ecc.

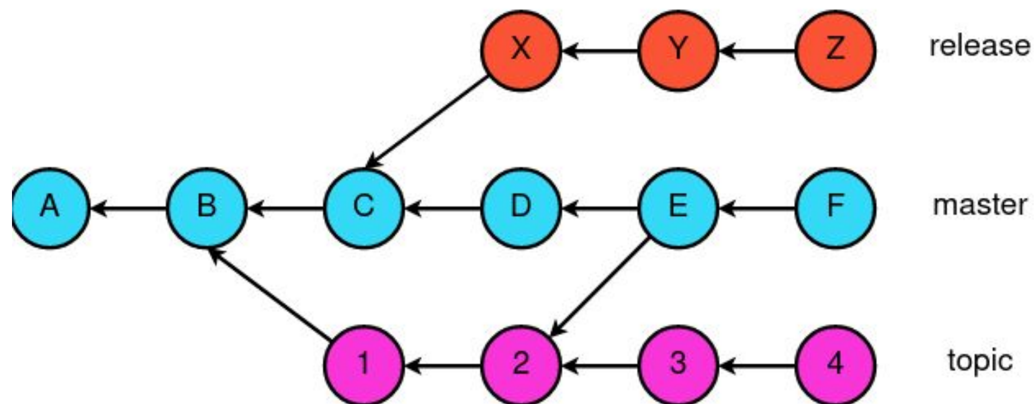
come funziona?

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

La storia del repository (*repo*) si compone di **singoli snapshot** del progetto detti **commits**. Insieme, i commits formano i vertici di un **grafo aciclico diretto** (o orientato). Ogni commit punta ai suoi parent commits senza formare mai circuiti chiusi.



come funziona?



lettere e numeri rappresentano i **commit**

commit senza genitore sono **root commit** (A)

un commit con genitori multipli è detto **merge commit** (E)

Le etichette a destra del grafico sono chiamati **branches**

L'ultimo commit di ogni branch è detto **tip** del branch

perchè?

Git funziona completamente offline, senza un daemon di sistema, e si connette a internet solo quando esplicitamente richiesto dall'utente.

Permette di lavorare in due contesti principali:

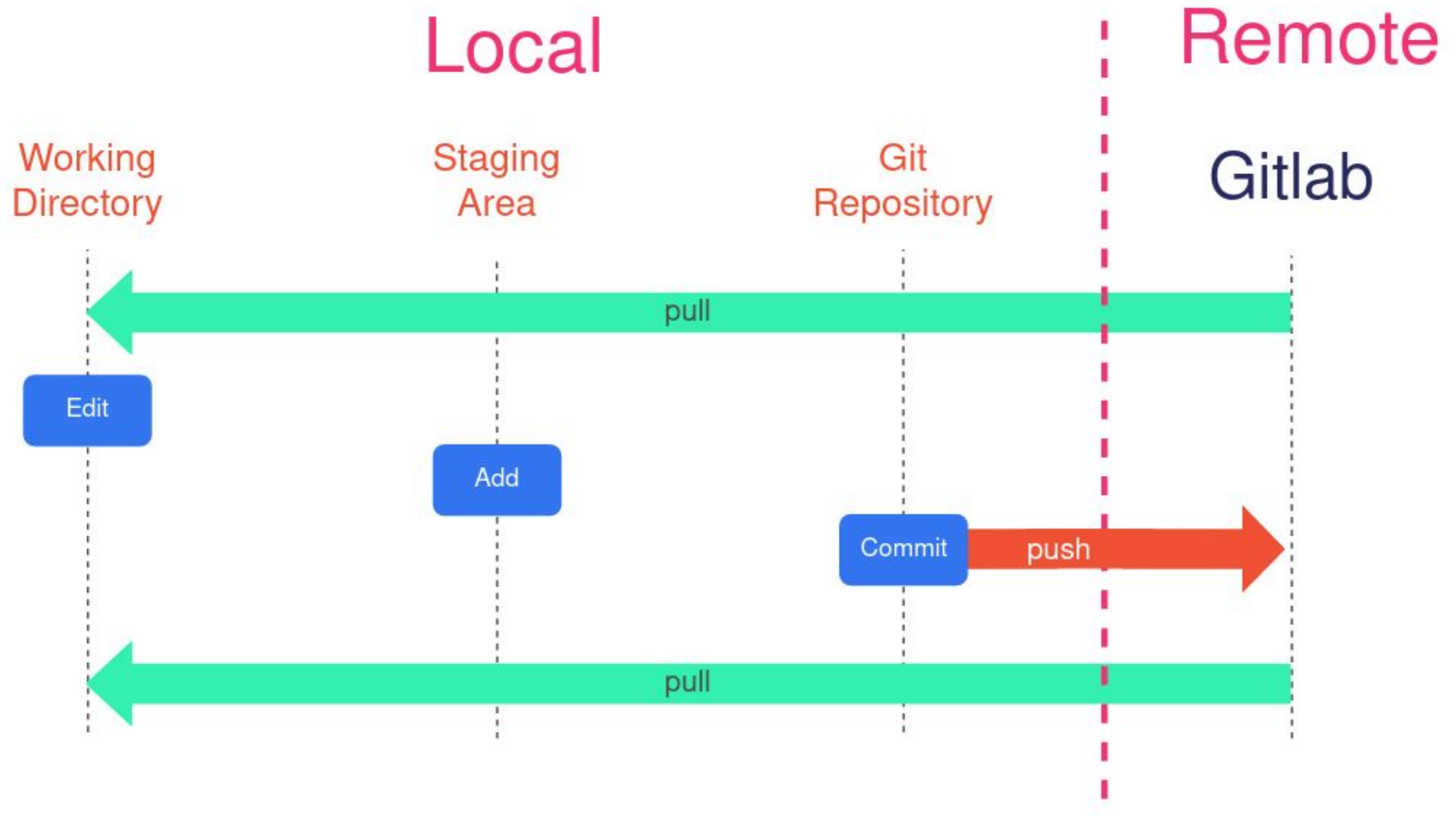
- **lavoro locale in privato**: commit frequenti, tanti branch, per avere la libertà di poter sperimentare senza la preoccupazione di perdere uno stato precedente o interferire con il lavoro degli altri

  → Comandi principali: **add, commit**

- **condivisione e pubblicazione**: condivisione di uno stato stabile del progetto come somma di uno o più commit locali

  → Comandi principali: **fetch, pull, push**

git workflow



git object store

Internamente, git utilizza solo **4 tipi di oggetti**, tutti memorizzati nel **git object store**:

- blob: insieme di byte senza una precisa struttura interna per quel che riguarda git. **Rappresenta il contenuto di un file**.

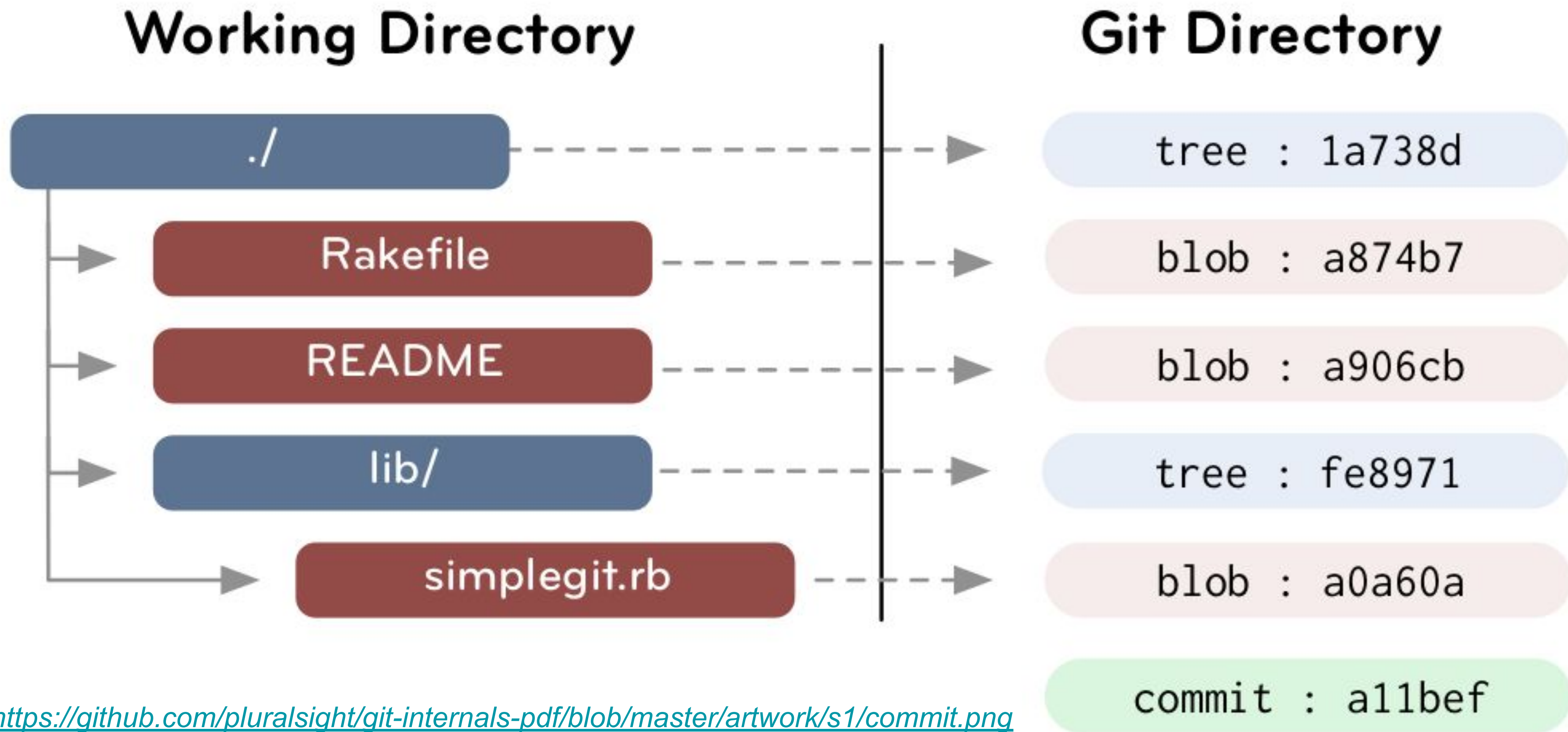
  Ogni versione di un file in git è rappresentata per intero in un blob separato. Successivamente, in fase di compressione, le parti ripetute vengono ottimizzate

- tree: lista di elementi presenti in un certo path. Ogni elemento può essere un blob (file) o un altro tree (sottocartelle).

- 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 object store



https://github.com/pluralsight/git-internals-pdf/blob/master/artwork/s1/commit.png

git object store

Gli oggetti git sono identificati tramite un **hash crittografico SHA1** calcolato in base al contenuto. Pertanto sono **immutabili**.

Gli oggetti si referenziano 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 7 caratteri** per riferirsi all'oggetto.

installazione

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

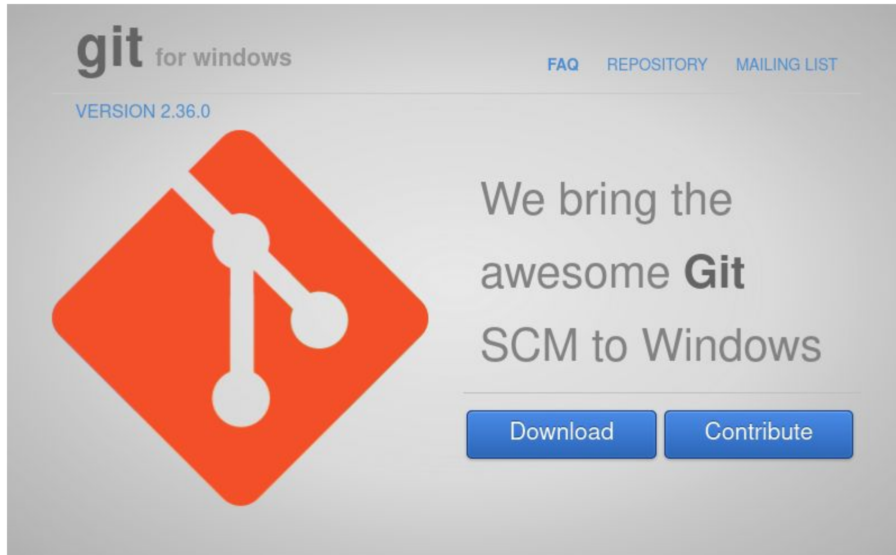
```
$ git help
usage: git [--version] [--help] [-C <path>]
           [-c <name>=<value>] [--exec-path[=<path>]]
           [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager]
           [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>]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

git su windows

Git for Windows https://gitforwindows.org/ offre un wizard di installazione guidato che permette di configurare fin da subito molte opzioni.

Inoltre, vengono installati anche:

- Git BASH: emulatore di bash per usare git da riga di comando.
- Git GUI: interfaccia grafica per utilizzare git. *Sconsigliato.*
- Integrazione con explorer: gli strumenti di git possono essere aperti direttamente nella cartella giusta dal menù di contesto (tasto destro in Windows Explorer)



git config

Prima di utilizzare `git` è necessario configurare alcuni parametri base con il comando:

```
git config --[local|global|system] <parameter> <value>
```

Git permette di specificare configurazioni a **3 livelli**, salvati in 3 file diversi:

- **local** → **.git/config** per il repository corrente (**default**)
- **global** → **~/.gitconfig** per l'utente corrente
- **system** → **/etc/gitconfig** per tutti gli utenti del sistema

Nel caso un'opzione fosse specificata a più di un livello, prevale quello più specifico:

```
local > global > system
```

git config - parametri -

I primi parametri che è opportuno configurare sono **`user.name`** e **`user.email`**: nome e indirizzo email con cui verrà salvato il proprio lavoro

```
$ git config --global user.name “Nome Cognome”
$ git config --global user.email “dev@nomecognome.it”
```

`git config --list`: mostra le configurazioni correnti

```
$ git config --list
user.email=dev@nomecognome.it
user.name=Nome Cognome
core.editor=vim
core-excludesfile=/home/davide/.gitignore
```

Newline e git

I sistemi Unix e Windows differiscono nella rappresentazione della nuova riga:

- **Unix** e Unix-like → **LF** (Line Feed)
- **Windows** → **CR LF** (Carriage Return e Line Feed)

La differenza può creare problemi nella condivisione del lavoro tra sistemi operativi.

Con l'impostazione

```
$ git config --global core.autocrlf true
```

git gestisce il problema in maniera trasparente.

Il carattere di Carriage Return viene rimosso dai file salvati da git, e temporaneamente reintrodotto solo quando i file vengono utilizzati su Windows.

L'impostazione è attivata di default dal wizard di installazione su Windows.

init

git init

Per poter utilizzare git bisogna **inizializzare il repository** che conterrà il nostro progetto:

`git init [directory]`

Il comando crea la *directory* se necessario. La *directory* di default è quella corrente `.`

All’interno della *directory* viene creata una cartella chiamata **`.git`**.

La cartella `.git` contiene tutti gli oggetti necessari al suo funzionamento.

Solitamente non è necessario modificarla in alcun modo.

```
$ mkdir tutorial
$ cd tutorial
$ git init
Initialised empty Git repository in
/home/davide/tutorial/.git/
```

git status

Il comando git status mostra lo **stato del repository**.

Il comando può essere utilizzato per ottenere un riepilogo di quali file sono stati modificati, su quale branch ci troviamo, e tante altre informazioni.

È buona norma utilizzare questo comando per tenere sotto controllo la situazione del proprio repository. Spesso fornisce anche **suggerimenti preziosi su come procedere**!

```
$ git status
On branch main
nothing to commit, working tree clean
```

Situazione ideale su un repository aggiornato

add & commit

git add

DIGITAL DREAMERS ASSOCIAZIONE

Per iniziare, bisogna indicare **quali file git deve tracciare** con `git add <nomefile>`. Se il file è già tracciato da git, verrà aggiunto solo in caso siano presenti cambiamenti.

Si può anche specificare **una cartella** per aggiungere tutto il suo contenuto ricorsivamente, ad esempio: `$ git add .`

1. Creo un file `$ touch first.txt`
2. Controllo lo stato del repo: un file è stato modificato, ma non è ancora tracciato

```
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    first.txt

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

git add

3. Aggiungo il file: `$ git add first.txt`

Lo stato del repository è cambiato:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   first.txt
```

Il file è ora tracciato da git, pronto per essere registrato in modo definitivo nella storia del progetto tramite un **commit**.

DIGITAL DREAMERS
ASSOCIAZIONE

commit

Per poter confermare le modifiche effettuate si utilizza il comando git commit.

Il commit è **un'istantanea dello stato del progetto** ed è **identificato univocamente da un hash crittografico SHA1** calcolato in base ai suoi contenuti:

- struttura di file e cartelle del progetto (*tree*)
- autore + data e ora
- commit message
- committer + data e ora
- hash di 0..N parent commits

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

Esiste almeno un commit nel repository senza parent: **root commit**.

Dato che ogni commit punta ai propri genitori, se due persone hanno lo stesso commit, hanno la certezza di avere la stessa storia.

commit

È obbligatorio accompagnare il commit con un **messaggio** di descrizione delle modifiche effettuate. Il messaggio può essere passato con il parametro **`-m "Messaggio"`**

4. Eseguo il commit del file aggiunto.

```
$ git commit -m "Added my first, empty file"
[main 9fca187] Added my first, empty file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 first.txt
```

È stato creato un commit sul branch main con hash **9fca187**. Rispetto alla versione precedente, è stato modificato un solo file, con 0 righe aggiunte o rimosse.

Se il parametro –m viene omesso, git apre l'editor di testo di default (configurabile) per permettere l'inserimento del messaggio.

add e commit

Per riassumere, il flusso di lavoro più lineare prevede di:

1. effettuare una serie di modifiche correlate
2. aggiungere le modifiche: **`git add`**
3. confermare le modifiche: **`git commit`**

Il comando add aggiunge i file modificati ad un'**area di staging**. Il comando commit registra solo i cambiamenti presenti nell'area di staging, ignorando altri eventuali modifiche ai file del progetto.

`git commit -a` (shorthand per `--all`) permette di includere nel commit tutti i file modificati e già tracciati nel repo, senza doverli aggiungere.

N.B.: file nuovi e mai aggiunti non verranno inclusi.

DIGITAL DREAMERS
ASSOCIAZIONE

add e commit

Effettuato un commit si ha un nuovo stato del progetto da cui partire per registrare ulteriori modifiche.

Untracked | Unmodified | Modified | Staged

Add the file

Edit the file

Stage the file

Remove the file

Commit

https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository

commit

Commit early, commit often!

I commit dovrebbero essere **piccoli**, **frequenti** e **atomici**: dovrebbero contenere tutti e soli i cambiamenti legati ad una certa funzionalità, o un'unità di lavoro nella sua interezza.

In questo modo, diventa più facile:

- ripercorrere la storia del progetto a posteriori
- comprendere i cambiamenti fatti
- rintracciare le origini di un bug
- eventualmente, **tornare a uno stato precedente**.

Idealmente, un commit dovrebbe portare l'applicazione da uno stato funzionale A ad uno stato funzionale B.

commit: messaggio

L'atomicità del commit si riflette anche nel **messaggio**:

- la prima riga del commento (per convenzione **max 50 o 80 caratteri**) è privilegiata perché spesso è l'unica che viene visualizzata: deve **riassumere i cambiamenti fatti**
- eventuali dettagli possono essere aggiunti nelle righe successive

Un messaggio di commit troppo lungo e articolato è un possibile segnale che i cambiamenti dovrebbero essere spezzati su più commit.

```
$ git commit -m "Add unit tests to login flow"

[main (root-commit) 548abd3] Fixed bug with first file
1 file changed, 1 insertion(+)
create mode 100644 first.txt
```

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary. In some contexts, the
first line is treated as the subject of an email and the rest of
the text as the body.

Write your commit message in the imperative: "Fix bug" and not
"Fixed bug" or "Fixes bug."

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet

If you use an issue tracker, add a reference(s) to them at the
bottom, like so:

Resolves: #123
```

Esempio di messaggio di commit, abbreviato:

https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html

	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
○	AAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

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

https://xkcd.com/1296/

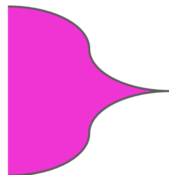
git rm

Cancellare un file è una modifica come le altre!

Si segue la stessa procedura add → commit.

In alternativa si può usare `git rm <nomefile>` per rimuovere il file e aggiungere subito la modifica in staging.

1. `$ rm trashfile.txt`
2. `$ git add trashfile.txt`
3. `$ git commit`



1. `$ git rm trashfile.txt`
2. `$ git commit`

git restore

`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 first.txt
$ git status
On branch main
nothing to commit, working tree clean
```

.gitignore

Spesso un progetto contiene file che non si vuole tracciare o condividere, ad esempio:

- file con password o altri dati sensibili
- file di configurazione diversi per ogni installazione
- file di cache o file compilati
- file di log
- file non attinenti al progetto, Es: `Thumbs.db` su Windows, `.DS_Store` su Mac, file generati dal proprio IDE o editor (cartelle `.idea` e `.vscode`)

Si possono specificare file e cartelle da omettere in un file di testo .gitignore.

.gitignore contiene l'elenco dei pattern da ignorare, uno per riga. Si possono utilizzare wildcard per creare pattern complessi.

.gitignore

Si trovano template preparati a seconda del linguaggio di programmazione utilizzato:
https://github.com/github/gitignore

```
# C - Precompiled Headers
*.gch
*.pch
```

```
# NodeJS - Logs
logs
*.log
npm-debug.log*
yarn-error.log*
```

```
# Python - Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class
```

esplorare la storia

git log

Esplorare la storia è utile per poter tenere monitorati tutti i cambiamenti effettuati nel nostro repository. Utilizzando il comando `git log` si possono vedere:

- ID commit (hash)
- Autore
- Data e ora
- Messaggio di commit

```
$ git log
commit 548abd375bbeaa3a88e7f95c8d2a38afee59d262
(HEAD -> main)
Author: <Nome Cognome> <dev@nomecognome.it>
Date:   Fri Apr 22 15:39:28 2022 +0200

    First commit
```

Con molti commit e branch, la visualizzazione di default può risultare scomoda: git log offre parecchie **opzioni per personalizzarne l’output**

git log

Una delle opzioni più usate è `git log --oneline`, che visualizza un commit per riga e abbrevia l'hash a 7 caratteri:

```
$ git log --oneline
e35412f (HEAD -> main) A third commit, even
d52f43f A second commit, for the example
548abd3 First commit
```

Inoltre, il risultato può essere filtrato in vari modi:

- `--max-count=2` mostra al massimo 2 commit
- `--author='name'` mostra i commit di un certo autore
- `--since='5 minutes ago'` e `--until='31/12/2022'` filtrano per data. Si possono usare date assolute o relative.

git log

Le opzioni di `git log` sono tantissime. Consulta la documentazione!

Se si trova un formato particolarmente efficace, invece di dover specificare tutte le opzioni ogni volta, è possibile creare un alias.

Gli **alias** si definiscono nella configurazione: sono nomi personalizzati per eseguire certi comandi. Per creare un alias hist:

```
$ git config --global alias.hist "log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short"
```

A questo punto, eseguendo `$ git hist`

in realtà verrà eseguito

```
$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```

git log

- `--pretty=format:'%h %ad | %s%d [%an]'` specifica la formattazione del messaggio usando dei placeholder, ad esempio:
    - %h: hash abbreviato
    - %ad: data dell'autore
    - %s: prima riga del messaggio di commit ("subject")
- `--graph` mostra un grafico della storia dei branch (non evidente nell'esempio, sono gli asterischi a sinistra)
- `--date=short` usa il formato breve per le date

```
$ git hist

* e35412f 2022-04-24 | A third commit, even (HEAD -> main) [Nome Cognome]
* d52f43f 2022-04-23 | A second commit, for the example [Nome Cognome]
* 548abd3 2022-04-22 | First commit [Nome Cognome]
```

git diff

Il comando `git diff` evidenzia le **differenze fra due set di dati**.

Senza specificare parametri, diff mostra la **differenza fra lo stato attuale del repository e i file presenti nell'area di staging** (l'ultimo commit più tutti i `git add`).

Modificando "first.txt" ed eseguendo il comando git diff vengono mostrate le righe eliminate - e aggiunte +.

```
$ git diff
diff --git a/first.txt b/first.txt
index 47a25d4..e1e9897 100644
--- a/first.txt
+++ b/first.txt
@@ -1 +1 @@
Unchanged line.

-Original line.
+Edited line.
+
+A new line.
(END)
```

N.B.: Una modifica equivale a una rimozione più un'aggiunta

```
   past, invalid
13 -updates were a significant source of errors when updating ownCloud.
14
15  FAQ
16  ---
17
18  Why Did ownCloud Add Code Signing?
19  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
20
21 -By supporting Code Signing we add another layer of security by
    ensuring that
22 -nobody other than authorized persons can push updates for
    applications, and
23 -ensuring proper upgrades.
24
25  Do We Lock Down ownCloud?
26  ^^^^^^^^^^^^^^^^^^^^^^^^^
27
28 -The ownCloud project is open source and always will be. We do not
    want to make
29 -it more difficult for our users to run ownCloud. Any code signing
    errors on
30 -upgrades will not prevent ownCloud from running, but will display a
    warning on
31 -the Admin page. For applications that are not tagged "Official" the
    code signing
32 -process is optional.
33
34  Not Open Source Anymore?
35  ^^^^^^^^^^^^^^^^^^^^^^^^
36
```

```
   when updating ownCloud.
13
14  FAQ
15  ---
16
17  Why Did ownCloud Add Code Signing?
18  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
19
20 +By supporting Code Signing we add another layer of security which
    ensures that
21 +nobody, other than authorized individuals, can push updates for
    applications.
22 +This ensures proper upgrades.
23
24  Do We Lock Down ownCloud?
25  ^^^^^^^^^^^^^^^^^^^^^^^^^
26
27 +The ownCloud project is open source and always will be.

28 +We do not want to make it more difficult for our users to run
    ownCloud.
29 +Any code signing errors on upgrades will not prevent ownCloud from
    running, but will display a warning on the Admin page.
30 +For applications that are not tagged "Official" the code signing
    process is optional.
31
32  Not Open Source Anymore?
33  ^^^^^^^^^^^^^^^^^^^^^^^^
34
```

diff mostrata da GitHub

git diff

Specificando i set di dati sui quali operare è possibile rilevare le differenze fra due branch, due file, due commit, due path, ecc…

Il comando funziona anche al di fuori di repository git!

Ad esempio: per mostrare le differenze che si sono accumulate nel progetto tra il primo e il terzo commit del repo:

```
$ git diff 548abd3..e35412f
```

(hash dei commit separati da doppio punto ..)

Per mostrare le differenze tra l'ultimo commit e l'area di staging (cioè tutto ciò che è stato aggiunto con git add):

```
$ git diff --staged
```

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 contente le operazioni opposte a quelle del commit target.

1. Si esegue il commit "errato"

```
$ git add wrong.txt
$ git commit -m "Oops, we didn't want this commit"
[main 8258137] Oops, we didn't want this commit
 1 file changed, 1 insertion(+)
```

2. Si ripristina lo stato precedente

```
$ git revert 8258137
[main 88bf9e7] Revert "Oops, we didn't want this commit"
 1 file changed, 1 deletion(-)
```

git revert

Al contrario di altre soluzioni, `git revert` **preserva la storia** del repository.

```
$ git hist
* 88bf9e7 2022-04-26 | Revert "Oops, we didn't want this commit"
(HEAD -> main) [Nome Cognome]
* 8258137 2022-04-26 | Oops, we didn't want this commit [Nome
Cognome]
* e35412f 2022-04-24 | A third commit, even [Nome Cognome]
* d52f43f 2022-04-23 | A second commit, for the example [Nome
Cognome]
* 548abd3 2022-04-22 | First commit [Nome Cognome]
```

DIGITAL DREAMERS
ASSOCIAZIONE

git reset

Il comando git reset <target> riporta il repository allo stato del target, ad esempio il nome di un branch o l'ID di un commit, scartando delle modifiche e a volte interi commit.

Il target di default è HEAD.

> Come la testina che legge un nastro, HEAD è il nome con cui git si riferisce al branch e al commit su cui il repository si trova in un dato momento.

`git reset` sposta solamente HEAD. Specificando l'opzione `--hard` anche i file verranno ripristinati allo stato precedente. ⚠️ *Attenzione! Possibile perdita di dati!* ⚠️

(Alcuni utilizzi di git reset si sovrappongono a quelli di restore e non vengono trattati.)

git reset

Nel suo utilizzo più semplice, `git reset` rappresenta l'inverso di `git add`: il repo torna allo stato in cui si trovava dopo il commit. Le modifiche vengono conservate e possono essere nuovamente aggiunte.

1. Si modificano e aggiungono dei file: `$ git add first.txt second.txt`

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   first.txt
    modified:   second.txt
```

git reset

`reset`

2. Si ripristina lo stato del repo: le modifiche vengono rimosse dall’area di staging ma rimangono nei file

```
$ git reset
Unstaged changes after reset:
M   first.txt
M   second.txt
```

`git status` mostrerà modifiche da aggiungere.

`reset --hard`

2. Si ripristina lo stato del repo e le modifiche vengono scartate anche dai file

```
$ git reset --hard
HEAD is now at 8258137 Revert
"Oops, we didn't want this
commit"
```

`git status` mostrerà un working tree pulito: le modifiche sono state perse.

git reset

Specificando l'ID di un commit, `git reset` riporta il repo in quello stato.

1. Si può tornare al primo commit: `$ git reset 548abd3`
2. `git log` mostra come **la storia sia stata persa** *(o quasi…)*

```
$ git log --oneline
548abd3 (HEAD -> main) First commit
```

3. `git status` e `git diff` mostrano tutte le modifiche aggiunte tra il primo e l'ultimo commit.

con l'opzione `--hard` *sarebbero state perse.*
`git status` *mostrerebbe un repository pulito.*

git reflog

Anche una volta dereferenziati, git tiene i suoi oggetti in memoria per qualche tempo prima di eliminarli. I commit scartati si possono recuperare tramite il reflog.

`git reflog` mostra la storia di dove si è mossa HEAD e perché. Può essere utile quando si crea confusione e si vuole tornare a uno stato noto.

4. Si recupera l’ID dell’ultimo commit

```
$ git reflog
548abd3 HEAD@{0}: reset: moving to 548abd3
88bf9e7 (HEAD -> main) HEAD@{1}: commit: Revert "Oops..."
8258137 HEAD@{2}: commit: "Oops..."
```

5.Si ripristina lo stato precedente `$ git reset 88bf9e7`

branching e merging

DIGITAL DREAMERS
ASSOCIAZIONE

branch

I branch sono **ramificazioni nella storia del repository**. Il primo branch viene creato insieme al repository stesso.

Si crea un nuovo branch per **isolare** il lavoro su una funzionalità dal resto del progetto, evitando di “sporcare” il ramo principale, e permettendo di lavorare facilmente in parallelo.

Little Feature

Main

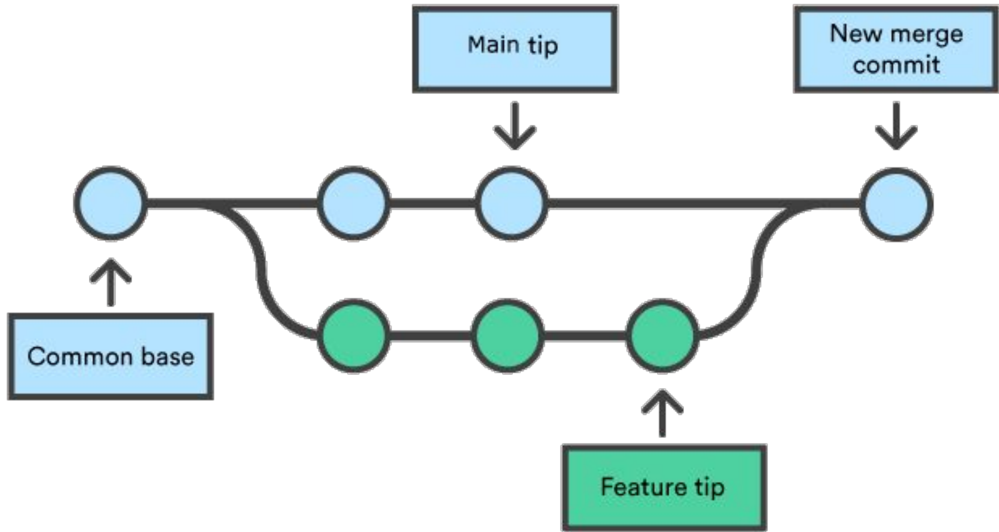
Big Feature

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

branch

Una volta pronto, un branch può essere riunito al ramo principale tramite l’operazione di merge.

Storicamente il branch principale prendeva di default il nome master ma da vari anni si preferisce il termine main. (configurabile)



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

DIGITAL DREAMERS
ASSOCIAZIONE

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.

branch

3. **Nuovi commit verranno aggiunti sul branch `my-feature`**

```
$ git commit -am "Add new feature"
[my-feature 71d740c] Add new feature
 1 file changed, 1 insertion(+)
```

4. Il branch `my-feature` è avanzato, mentre `main` è rimasto al commit precedente.

```
$ git log --oneline
71d740c (HEAD -> my-feature) Add new feature
88bf9e7 (main) Revert "Oops, we didn't want this commit"
8258137 Oops, we didn't want this commit
```

Tornando sul branch `main` `$ git switch main` le modifiche effettuate su `my-feature` non saranno visibili.

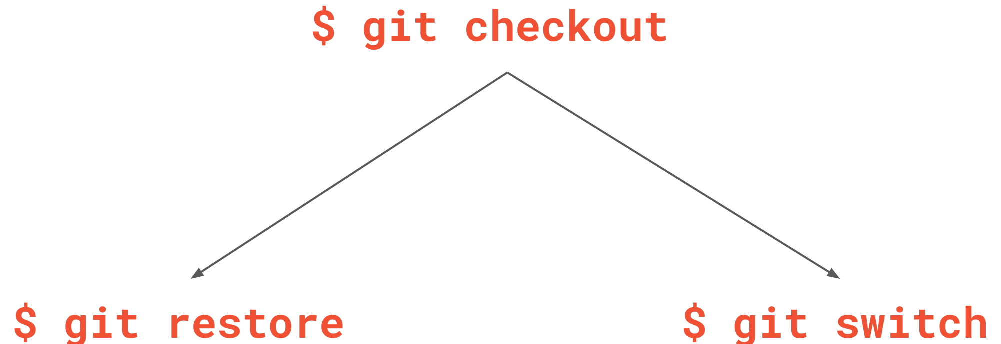
git checkout

`git switch` è un comando recente, introdotto con la versione 2.23 del 08/2019.

Storicamente, il comando per cambiare branch è stato `git checkout <name>`, a cui aggiungere l'opzione `-b` per creare un nuovo branch.

`git checkout` è tuttora supportato, e si trova ancora spesso, ma non è più consigliato per motivi di chiarezza: nel tempo **ha accumulato troppe funzionalità**, creando confusione.

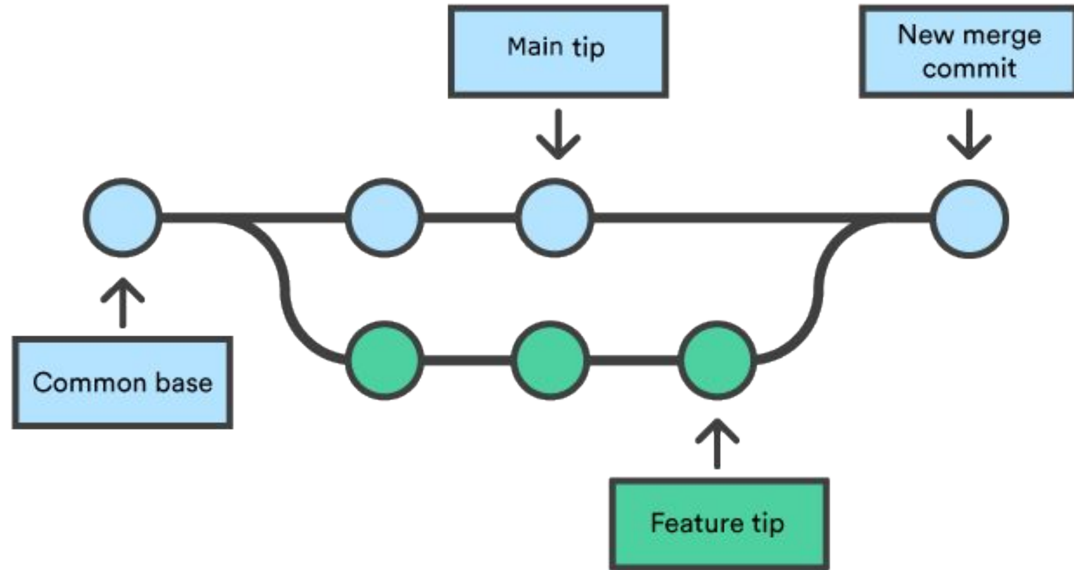
I due usi descritti dalla documentazione (*"Switch branches or restore working tree files"*) sono stati reimplementati in comandi distinti.



git merge

Per portare i cambiamenti da un branch all'altro si usa il comando git merge. Se necessario git crea un "merge commit" per riconciliare le storie dei branch.

```
$ git switch main
$ git merge feature
```

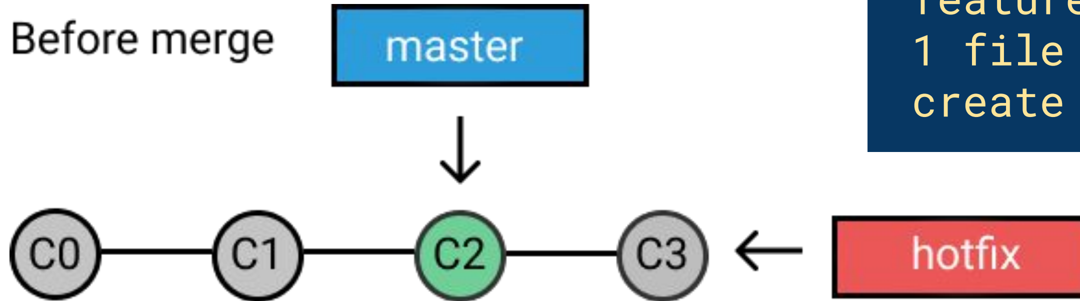


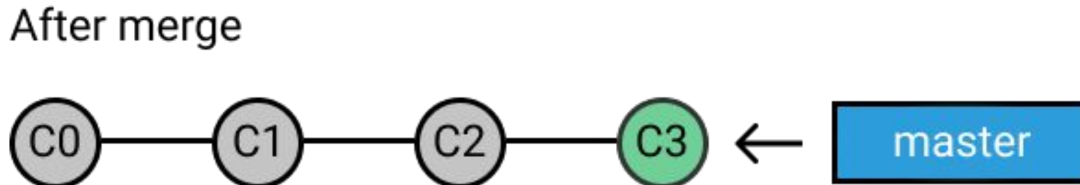
https://www.atlassian.com/it/git/tutorials/using-branches/git-merge

git merge

Esempio di merge in fast-forward, ovvero senza "merge commit"

```
$ git switch main
Switched to branch 'main'
$ git merge my-feature
Updating 88bf9e7..71d740c
Fast-forward
 feature.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
```





https://kolosek.com/git-merge/

workflow con branches

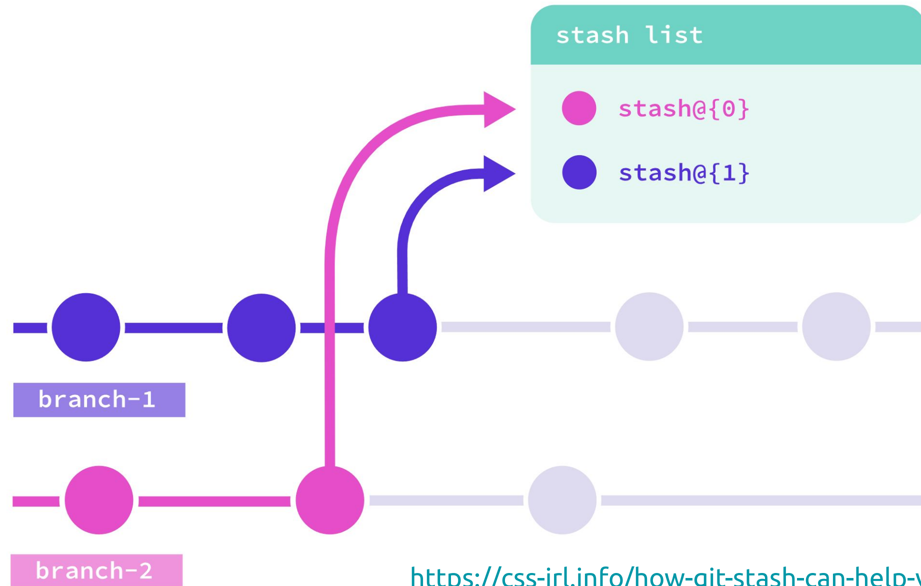
Il workflow più semplice con git prevede di:

- Avere (almeno) **un branch di riferimento**, di solito main.
    - spesso main rappresenta la versione del software attualmente in produzione
    - si possono avere altri branch per le versioni del software in beta, alpha ecc.
- Aprire un **nuovo branch** (basato sul branch di riferimento) quando si vuole lavorare su una nuova funzionalità.
    - Avere branch separati isola il proprio lavoro, e permette lo sviluppo in parallelo
        - Ultimato il lavoro sul branch, riportare i cambiamenti sul branch di riferimento tramite l'**operazione di merge**.

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.



https://css-irl.info/how-git-stash-can-help-you-juggle-multiple-branches/

git stash

1. Mentre si lavora sul branch `my-feature`, viene chiesto di fare una modifica su `another-feature`. Le modifiche parziali non sono pronte per un commit, quindi vengono “messe da parte”

```
$ git stash push
Saved working directory and index state
WIP on my-feature: 71d740c Add new feature
$ git switch another-feature
```

2. Si lavora sul branch `another-feature`, ma viene scoperto un bug urgente su `main`. Ancora una volta si mette da parte il proprio lavoro:

```
$ git stash
$ git switch main
```

Il comando push è il default e può essere omesso

git stash

3. Si possono consultare tutte le modifiche messe da parte, con la più recente in alto:

```
$ git stash list
stash@{0}: WIP on another-feature: 32bc3fa Work off-screen
stash@{1}: WIP on my-feature: 71d740c Add new feature
```

4. Si possono **riapplicare le modifiche** (anche su un branch diverso da quello originale)

```
$ git stash apply stash@{0}
[output di git status]
```

5. Le modifiche salvate possono essere **eliminate**:

```
$ git stash drop
Dropped stash@{0} (dcfca54...)
```

gestire i conflitti

DIGITAL DREAMERS
ASSOCIAZIONE

Cos'è un conflitto?

In git, un conflitto si presenta durante operazioni come merge o rebase quando due o più modifiche concorrenti sono state applicate sulla stessa riga di codice.

In caso di conflitto git sospende l'operazione e chiede all'utente di risolvere la situazione.

Simuliamo un conflitto dovuto al lavoro in parallelo di due developers:

1. A partire da main, si crea un nuovo branch: `$ git switch -c some-work`
2. Si aggiunge una riga in cima a first.txt e si fa il commit
3. Si torna su main: `$ git switch main`
4. Si aggiunge una riga in cima a first.txt (con un contenuto diverso dalla precedente) e si fa il commit
5. Si tenta di fare il merge: `$ git merge some-work`

Cos'è un conflitto?

Git non riesce a eseguire il merge:

```
$ git merge some-work
Auto-merging first.txt
CONFLICT (content): Merge conflict in first.txt
Automatic merge failed; fix conflicts and then commit the result.
```

`git status` segnala che è necessario un merge commit, e suggerisce come procedere:

```
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:   first.txt
```

Gestire i conflitti

Git ha marcato il file fonte di conflitti.

L'inizio del conflitto è segnato con `<<<<<<< HEAD` e continua fino a `=======`. Qui si trova la riga secondo HEAD (quindi il branch corrente).

```
<<<<<<< HEAD
Line added in main
=======
Line added in some-work
>>>>>>> some-work
```

first.txt

Da `=======` fino a `>>>>>>> some-work` invece si trova la riga introdotta dal branch.

Per procedere, è sufficiente seguire le indicazioni fornite da `git status`:

- modificare il file in modo da ottenere il risultato corretto
- `git add first.txt` → `git commit`
  (N.B: i merge commit suggeriscono un messaggio di default.)

remotes

remotes

Git nasce come **sistema distribuito** per facilitare il **lavoro in team**. A questo scopo 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.

La comunicazione e l'autenticazione possono avvenire tramite vari protocolli.

Protocollo	Autenticazione	Note
https	username e password, solo quando necessario	Più semplice per l'utente
ssh	chiave pubblica-privata, sempre	Più sicuro
filesystem	Permessi di lettura e scrittura del FS	Raro

remotes

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

```
$ git remote add origin https://gitlab.com/ddreamers/my-project.git
```

oppure con ssh:

```
git@gitlab.com:ddreamers/my-project.git
```

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

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

git clone

È raro aggiungere remotes in corso d’opera. Spesso, si contribuisce a **progetti già avviati**.

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

1. Si esce dal progetto: `$ cd ..`
2. 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:

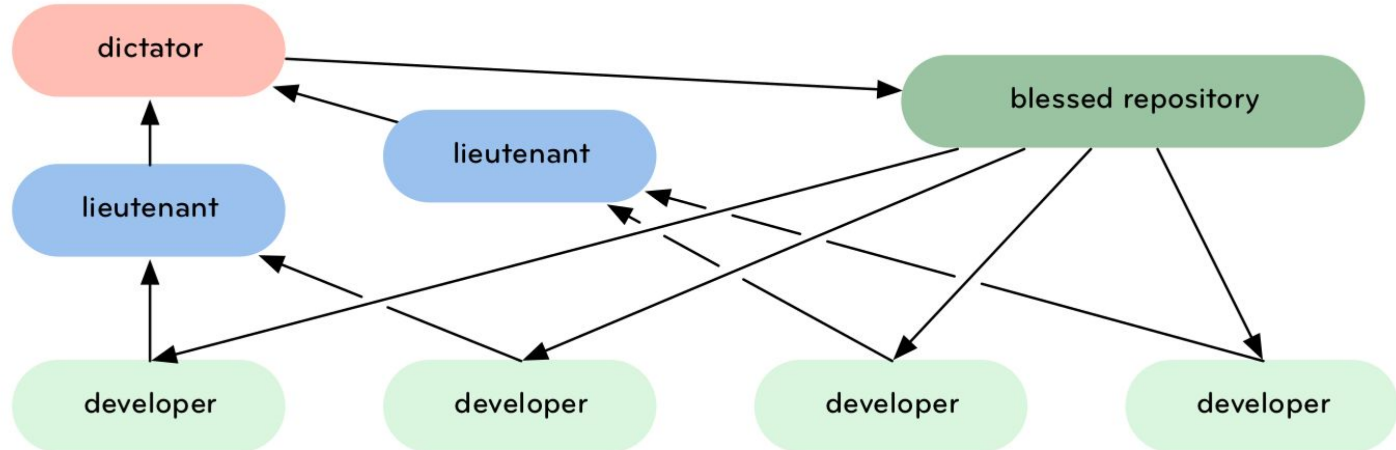
```
$ git clone tutorial learning_remotes
Cloning into 'learning_remotes'...
done.
```

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.

workflow

Essendo un **sistema distribuito**, git non impone un particolare modello di lavoro. Spesso si sceglie di lavorare con un repo centrale condiviso a cui fare riferimento, ma a livello architetturale **ogni copia di un repo ha pari dignità**.

Ad esempio, il kernel Linux viene sviluppato secondo un **modello stratificato**, in cui il lavoro di un team viene progressivamente approvato a livelli più alti fino alla pubblicazione.



https://github.com/pluralsight/git-internals-pdf/blob/master/artwork/s2/workflow-dictator.png

DIGITAL DREAMERS
ASSOCIAZIONE

workflow

L'**organizzazione a stella** segue il modello client/server. È il flusso di lavoro più comune, e architetturalmente coerente con l'utilizzo di servizi di **hosting centralizzati** (Github/Gitlab)

developer
developer
developer
developer
shared repository
developer
developer
developer
developer

https://github.com/pluralsight/git-internals-pdf/blob/master/artwork/s2/workflow-star.png

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**:

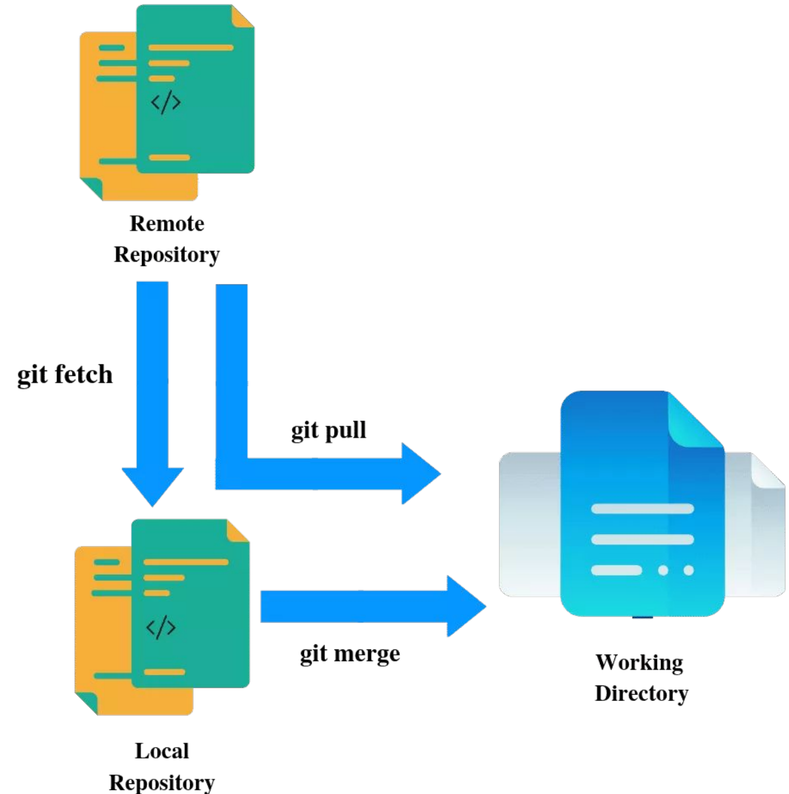
- con `git fetch` o `git pull` il repo locale **riceve** dal remote tutti i branch e i commit
- con `git push` si **invia** al remote il branch e il commit corrente e tutti i suoi antenati

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.



https://www.toolsqa.com/git/git-fetch-and-git-pull/

remotes

Un branch locale può “**tracciare**” un branch remoto. In moltissimi casi, l’associazione viene fatta in automatico, specie se branch locale e remoto hanno lo stesso nome.

1. Il repository `tutorial` (T) è il remote “origin” del nuovo `learning_remotes` (LR). L’intera storia di T, compresi tutti i branch, è nota a LR. I branch di LR tracciano i corrispettivi di T.

LR

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.
```

```
$ git branch -vv
* main 71d740c [origin/main] Add new feature
```

git fetch

2. Si esegue un commit sul branch main di T.

3. LR ancora non conosce il nuovo commit: `git status` non è cambiato. Per aggiornare lo stato del remote, si esegue `git fetch`:

LR

```
$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), 308 bytes | 308.00 KiB/s, done.
From /home/davide/tutorial
   71d740c..6cd18bd  main       -> origin/main
```

git pull

Anche `git status` segnala che il branch è indietro di un commit rispetto al remote:

```
$ git status
On branch main
Your branch is behind 'origin/main by 1 commit, and
can be fast-forwarded.
  (use "git pull" to update your local branch)
```

L'indicazione sul fast-forward significa che le storie dei branch **non sono divergenti**. Viene suggerito un `git pull` per aggiornare il branch locale con le modifiche del remote.

```
$ git pull
Updating 71d740c..6cd18bd
Fast-forward
 first.txt | 1 +
 1 file changed, 1 insertion(+)
```

git pull

Se le storie dei due branch fossero state 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

git push

Quando si fanno cambiamenti e si vogliono portare sul remote, si utilizza `git push`

1. Si crea un nuovo branch su cui lavorare: LR `$ git switch -c my-great-idea`
2. Si esegue un commit con delle semplici modifiche. Se si tenta di fare `git push`, appare un messaggio d'errore

LR

```
$ git push
fatal: The current branch my-great-idea has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin my-great-idea
```

Il nuovo branch non ha un corrispondente sul remote, e git chiede di impostarlo esplicitamente.

git push

3. Seguendo le istruzioni, si specifica il nome del branch sul remote. Per comodità si usa lo stesso nome che si ha in locale….

LR

```
$ git push --set-upstream origin my-great-idea
[...]
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To /home/davide/tutorial
 * [new branch]      my-great-idea -> my-great-idea
branch 'my-great-idea' set up to track 'origin/my-great-idea'.
```

Si può verificare che il branch è stato creato anche sul remote.

Specificare l'upstream è necessario solo la prima volta che si esegue il push di un nuovo branch.

commit e push

Commit early, commit often!

Push when necessary.

I commit devono essere **piccoli, frequenti e atomici**. Ogni commit deve contenere un set di modifiche coerenti tra loro, anche se è solo una riga di codice.

Se, scrivendo una funzione, si nota un problema non correlato, anche solo una formattazione del codice da correggere, si faranno due commit separati.

Fare il push dei propri commit può essere fatto **quando necessario**, per condividere le proprie modifiche, ottenere feedback, o anche solo come **forma di backup** alla fine di una giornata di lavoro.

Evitare push troppo frequenti su branch condivisi!

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.

DIGITAL DREAMERS ASSOCIAZIONE

git commit --amend

`git commit --amend` permette di **sostituire l'ultimo commit**, unendo nuove modifiche a quelle già contenute nel commit precedente. *(amend significa emendare, correggere)*

Se non si è aggiunto nulla all'**area di staging**, si può comunque modificare il messaggio.

Particolarmente utile per correggere "errori di distrazione". Invece di avere due commit…

```
$ git log --oneline
43dd8a1 forgot to add html files for login
f3b5449 Add login flow
```

…si può correggere la dimenticanza aggiungendo i file e eseguendo `git commit --amend`. Il risultato sarà un solo commit con nuovo hash:

```
$ git log --oneline
54c42df Add login flow
```

DIGITAL DREAMERS
ASSOCIAZIONE

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.

Feature
Feature
Main
Brand New Commits

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.

DIGITAL DREAMERS
ASSOCIAZIONE

git rebase interattivo

Il parametro `-i` (`--interactive`) espande enormemente le funzionalità di `git rebase`.

Viene presentata una lista dei commit da riapplicare, e per ognuno si può eseguire una determinata operazione, il tutto accompagnato da documentazione chiara e dettagliata.

```
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
```

Tra le opzioni, si segnala `squash`, che permette di unire un commit con il precedente, in modo analogo a `git commit --amend`. Infatti, al fine di mantenere una storia pulita, molti progetti FLOSS richiedono di appiattire le modifiche in un solo commit prima di integrarle nel branch principale.

DIGITAL DREAMERS
ASSOCIAZIONE

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.

git push --force-with-lease

Un'alternativa più sicura: `git push --force-with-lease`

Prima di eseguire la stessa operazione del semplice `--force`, controlla che non siano avvenuti cambiamenti a noi ancora sconosciuti sul remote.

Il comando verifica che **l'ultimo commit del branch remoto** che si sta per sovrascrivere sia quello che ci si aspetta (in base alle ultime operazioni di `fetch` o `push`).

In caso contrario, probabilmente qualcuno ha condiviso il proprio lavoro su quel branch, e l'operazione di `push` viene annullata per evitare di perdere i commit altrui.

⚠️ *Non è efficace in caso si utilizzino IDE o altri programmi che eseguono operazioni di* `git fetch` *periodicamente nel background!* ⚠️

cheatsheets

Install

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

Git for All Platforms
http://git-scm.com

Git distributions for Linux and POSIX systems are available on the official Git SCM web site.

Configure tooling

Configure user information for all local repositories

```
$ git config --global user.name "[name]"
```
Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```
Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```
Enables helpful colorization of command line output

Create repositories

When starting out with a new repository, you only need to do it once; either locally, then push to GitHub, or by cloning an existing repository.

```
$ git init
```
Turn an existing directory into a git repository

```
$ git clone [url]
```
Clone (download) a repository that already exists on GitHub, including all of the files, branches, and commits

The .gitignore file

Sometimes it may be a good idea to exclude files from being tracked with Git. This is typically done in a special file named `.gitignore` . You can find helpful templates for `.gitignore` files at github.com/github/gitignore.

Git Cheat Sheet by GitHub:
https://education.github.com/git-cheat-sheet-education.pdf
https://training.github.com/downloads/github-git-cheat-sheet.pdf

Git Cheat Sheet

GitLab

01 Git configuration

```
$ git config --global user.name "Your Name"
```

Set the name that will be attached to your commits and tags.

```
$ git config --global user.email "you@example.com"
```

Set the e-mail address that will be attached to your commits and tags.

```
$ git config --global color.ui auto
```

Enable some colorization of Git output.

02 Starting A Project

```
$ git init [project name]
```

Create a new local repository. If **[project name]** is provided, Git will create a new directory name **[project name]** and will initialize a repository inside it. If **[project name]** is not provided, then a new repository is initialized in the current directory.

```
$ git clone [project url]
```

Downloads a project with the entire history from the remote repository.

03 Day-To-Day Work

```
$ git status
```

Displays the status of your working directory. Options include new, staged, and modified files. It will retrieve branch name, current commit identifier, and changes pending commit.

```
$ git add [file]
```

Add a file to the **staging** area. Use in place of the full file path to add all changed files from the **current directory** down into the **directory tree**.

```
$ git diff [file]
```

Show changes between **working directory** and **staging area**.

```
$ git diff --staged [file]
```

Shows any changes between the **staging area** and the **repository**.

```
$ git checkout -- [file]
```

Discard changes in **working directory**. This operation is **unrecoverable**.

```
$ git reset [file]
```

Revert your **repository** to a previous known working state.

```
$ git commit
```

Create a new **commit** from changes added to the **staging area**.
The **commit** must have a message!

Git Cheat Sheet by GitLab:
https://about.gitlab.com/images/press/git-cheat-sheet.pdf

Git Cheat Sheet

GIT BASICS

Command	Description
git init <directory>	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
git clone <repo>	Clone repo located at <repo> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
git config user.name <name>	Define author name to be used for all commits in current repo. Devs commonly use --global flag to set config options for current user.
git add <directory>	Stage all changes in <directory> for the next commit. Replace <directory> with a <file> to change a specific file.
git commit -m "<message>"	Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
git status	List which files are staged, unstaged, and untracked.
git log	Display the entire commit history using the default format. For customization see additional options.

REWRITING GIT HISTORY

Command	Description
git commit --amend	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
git rebase <base>	Rebase the current branch onto <base>. <base> can be a commit ID, branch name, a tag, or a relative reference to HEAD.
git reflog	Show a log of changes to the local repository's HEAD. Add --relative-date flag to show date info or --all to show all refs.

GIT BRANCHES

Command	Description
git branch	List all of the branches in your repo. Add a <branch> argument to create a new branch with the name <branch>.
git checkout -b <branch>	Create and check out a new branch named <branch>. Drop the -b flag to checkout an existing branch.
git merge <branch>	Merge <branch> into the current branch.

Git Cheat Sheet by Atlassian:
https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

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

git

info@associazionedigitaldreamers.it