

Git

- 1) Come funziona Git
- 2) Il modello di storage
- 3) Staging Area
- 4) Branching
- 5) Presentazione dei comandi basilari
- 6) Sperimentazione

Come Funziona Git

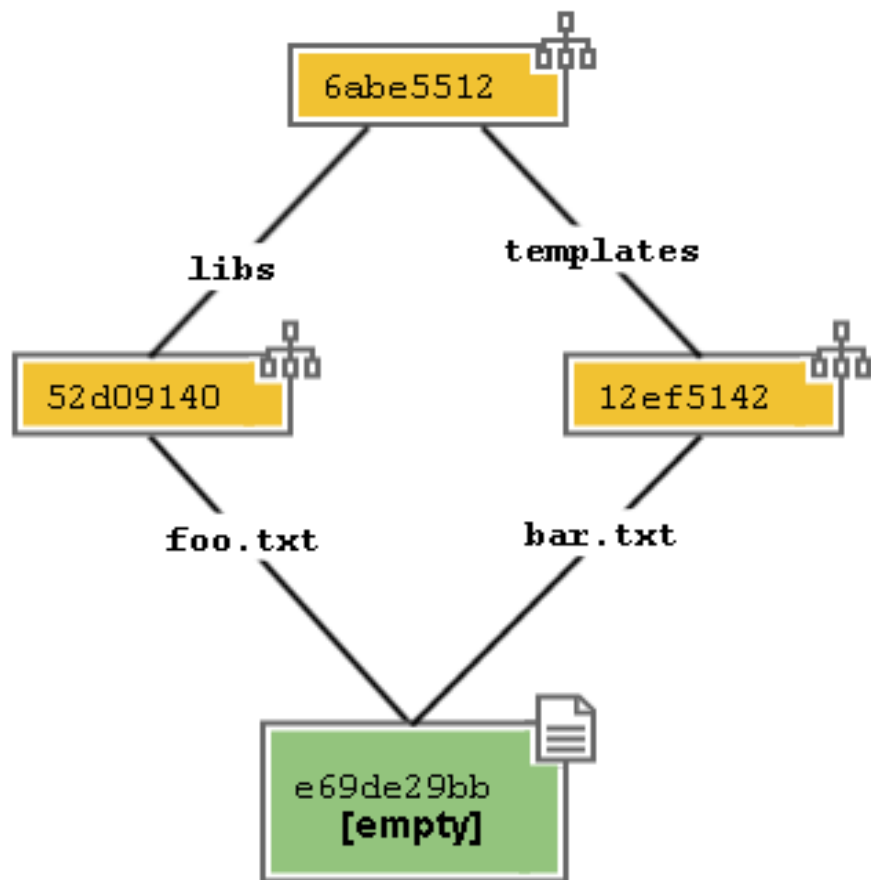
- Non c'è un server: il repository è locale. La gran parte delle operazioni è locale e non richiede l'accesso alla rete. Anche per questo git è incredibilmente veloce. (Riesce ad interagire con dei server remoti. Nonostante questo, resta sostanzialmente un sistema locale)
- Il progetto è indivisibile: git lavora sempre con l'intero codice sorgente del progetto e non su singole directory o su singoli file
- git non memorizza i cambiamenti dei file: git salva sempre i file nella loro interezza

Ottieni una copia locale dell'intero repository con tutta la storia del progetto e lavori in locale

Il modello di Storage (1)

- git memorizza i file così come sono, nella loro interezza; all'occorrenza ne calcola i delta di differenza.
- I file vengono memorizzati nel suo database chiave/valore, chiamato Object Database e conservato su file system nella directory nascosta .git. Nell'Object Database git ha memorizzato solo il contenuto del file, non il suo nome né la sua posizione
- nell'Object Database, git memorizza anche altri oggetti, chiamati tree che servono proprio a memorizzare il contenuto delle varie directory e i nomi dei file.

Il modello di Storage (2)



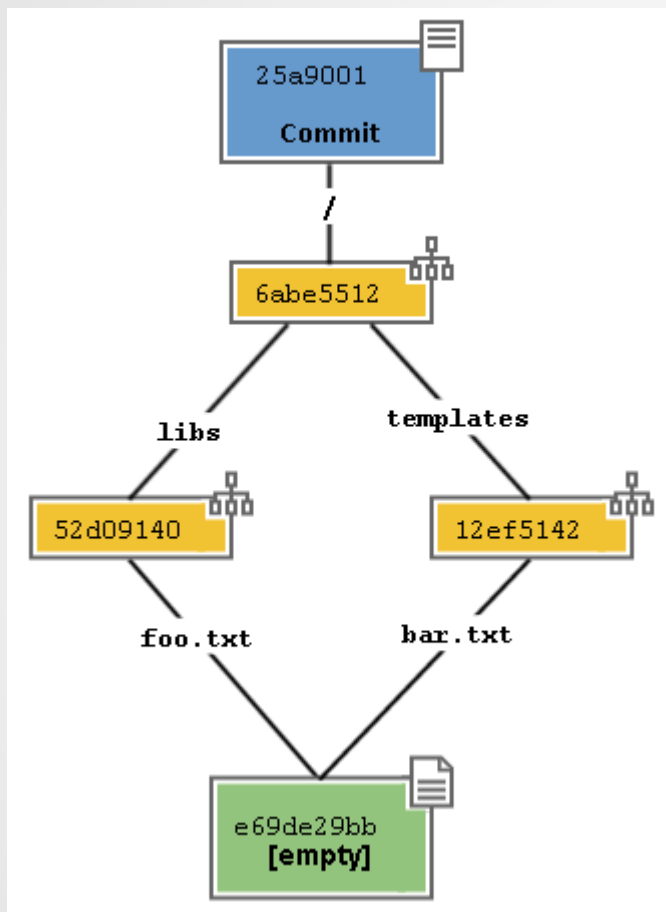
Root del progetto

Libs e templates sono 2 sotto-directory

foo.txt, bar.txt sono 2 file all'interno delle rispettive directories, puntano la stessa chiave in quanto sono tutti e 2 file vuoti

La Chiave è lo SHA1 del contenuto del file (e quindi univoco).
Il valore è il contenuto stesso del file.

Il modello di Storage (3)



Tutte queste strutture vengono raccolte dentro un contenitore, chiamato commit.

Il Commit è l'attuale fotografia del file System

Un Commit non è altro che un elemento del database chiave/valore, la cui chiave è uno SHA1, come per tutti gli altri oggetti, e il cui valore è un puntatore al tree del progetto, cioè la sua chiave (più un altro po' di informazioni, come la data di creazione, il commento e l'autore)

Staging area (Index) (1)

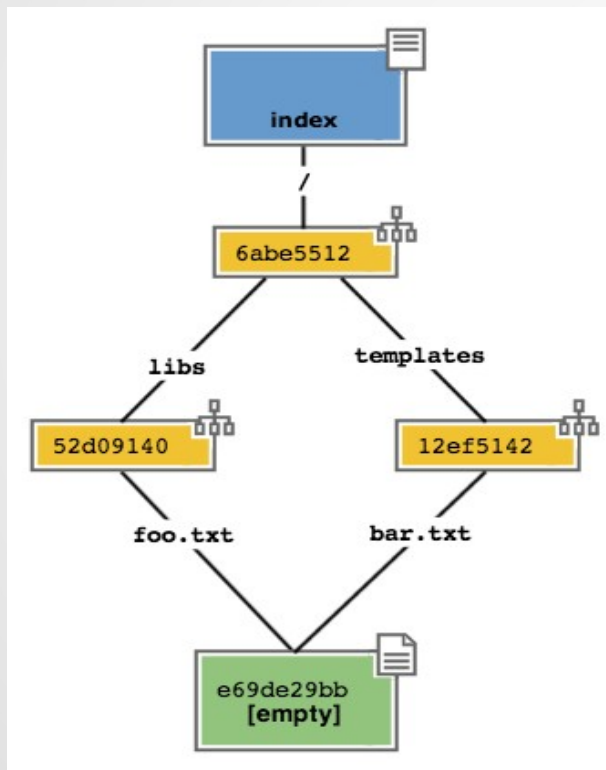
- L'index è una struttura che fa da cuscinetto tra il file system e il repository. È un piccolo buffer che puoi utilizzare per costruire il prossimo commit.



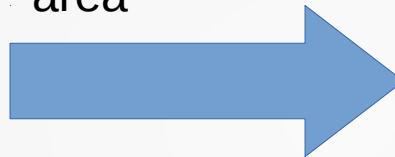
- 1) Il file system è la directory con i tuoi file.
- 2) Il repository è il database locale su file che conserva i vari commit
- 3) L'index è lo spazio che git ti mette a disposizione per creare il tuo prossimo commit prima di registrarlo definitivamente nel repository

Staging area (Index) (2)

L'index conserva una copia del tuo ultimo commit e si aspetta che tu lo modifichi.



Aggiunta e modifica di un file nel index area

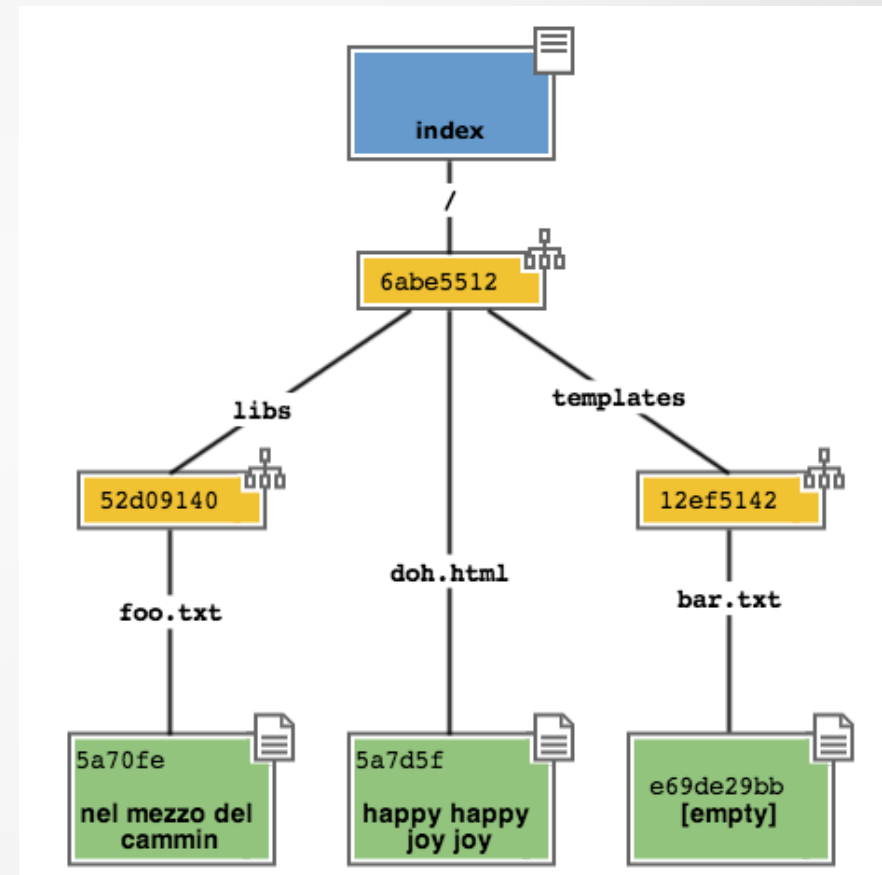


Comandi:

Git add doh.html

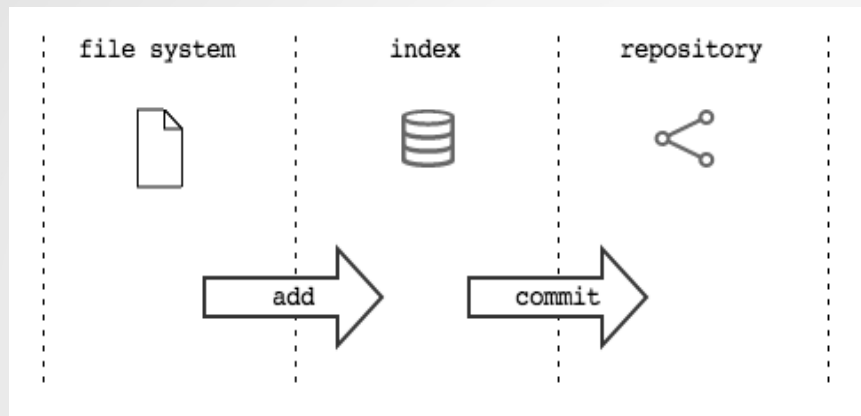
Modifico file di testo foo

Git add foo.txt



L'operazione add in git serve a salvare un file dentro l'index ed è un'operazione che va ripetuta ad ogni commit

Staging area (Index) (3)

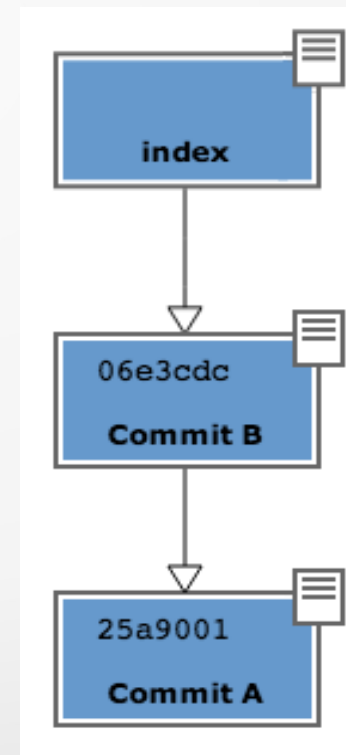


Quando tutte le modifiche sono state apportate al file, il commit non fa nient'altro che prendere l'oggetto index e andarlo a salvare nel database in modo opportuno. Successivamente l'index viene rilasciato e quest'ultimo punterà l'ultimo commit fatto.

Ogni oggetto commit ha un puntatore al commit padre da cui deriva.

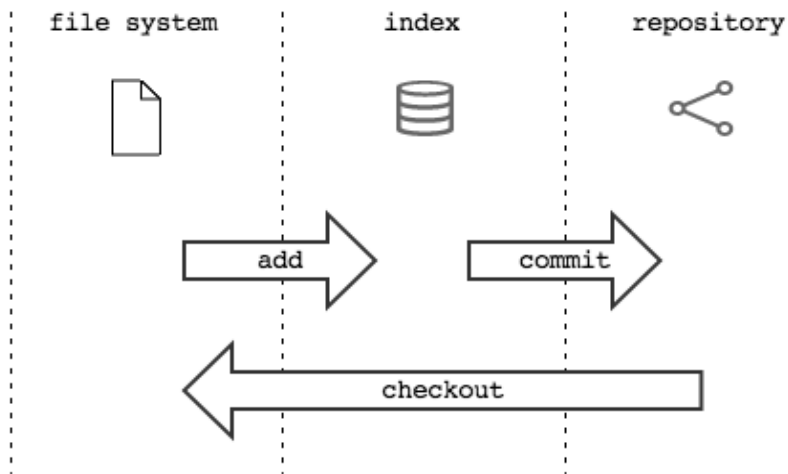
Per cui i commit sono esattamente la fotografia del file system in un dato momento.

Quindi posso ripercorrere la storia del progetto seguendo i puntatori ai commit padri.



Branching (1)

- Il progetto in git non segue uno sviluppo lineare. Infatti è possibile tornare indietro nel tempo (commit precedente).
- Essendo un database chiave/valore ogni commit può essere referenziato e richiamato.



Il comando checkout prende il commit indicato e lo copia nel file system e nella staging area. (Ripropone il progetto com'era al momento del commit).

Usando una rappresentazione grafica dei commit

A—B

Cosa succederebbe se con checkout andassimo sul commit A e da lì facessimo un commit C?

Si creerebbe una diramazione

A---B
\
C

Branching (2)

Gestire i commit A, B e C dovendoli chiamare con i propri SHA1 è di una scomodità unica.

Git risolve il problema permettendo di usare delle variabili per conservarne il valore dello SHA1 dei file.

Branch

Un branch in git non è un ramo ma bensì un'etichetta, un puntatore ad un commit, una variabile che contiene la chiave di un commit.

I branch vengono normalmente usati proprio per dare un nome ai rami di sviluppo.

Installare Git (1)

- Linux:
 - Per distribuzioni basate su Debian (come Ubuntu) utilizzare il semplice comando: `sudo apt-get install git`
 - Per altre distribuzioni (come Fedora) si può installare attraverso lo strumento base di amministrazione dei pacchetti della distribuzione. Esempio per Fedora: `yum install git`
- Windows:
 - Scaricare l'eseguibile da <http://msysgit.github.io/> e *lanciarlo (lasciare le impostazioni di default e se si vuole creare un collegamento sul Desktop).*

Installare Git (2)

- Mac:
 - Utilizzare l'installer grafico di git scaricabile da
<http://sourceforge.net/projects/git-osx-installer/>
 - Usando MacPorts(se già presente sulla macchina):
sudo port install git +svn +doc +bash_completion +gitweb
 - Usando HomeBrew(se già presente sulla macchina):
brew install git

Configurare git in remoto

Per evitare che una volta creata la repository remota i commit inviati su di essa siano “anonimi” e necessario configurare git .

Per farlo usiamo questi 2 comandi.

- 1) `git config --global user.name "NOME UTENTE"`
- 2) `git config --global user.email "EMAIL UTENTE"`

Comandi essenziali git in remoto (1)

- `git remote add [NOME] "URL"`
- `git clone "URL"`
- `Git fetch "NOME Server" "Branch"`
- `git pull`
- `git push "NOME Server" "Branch"`