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

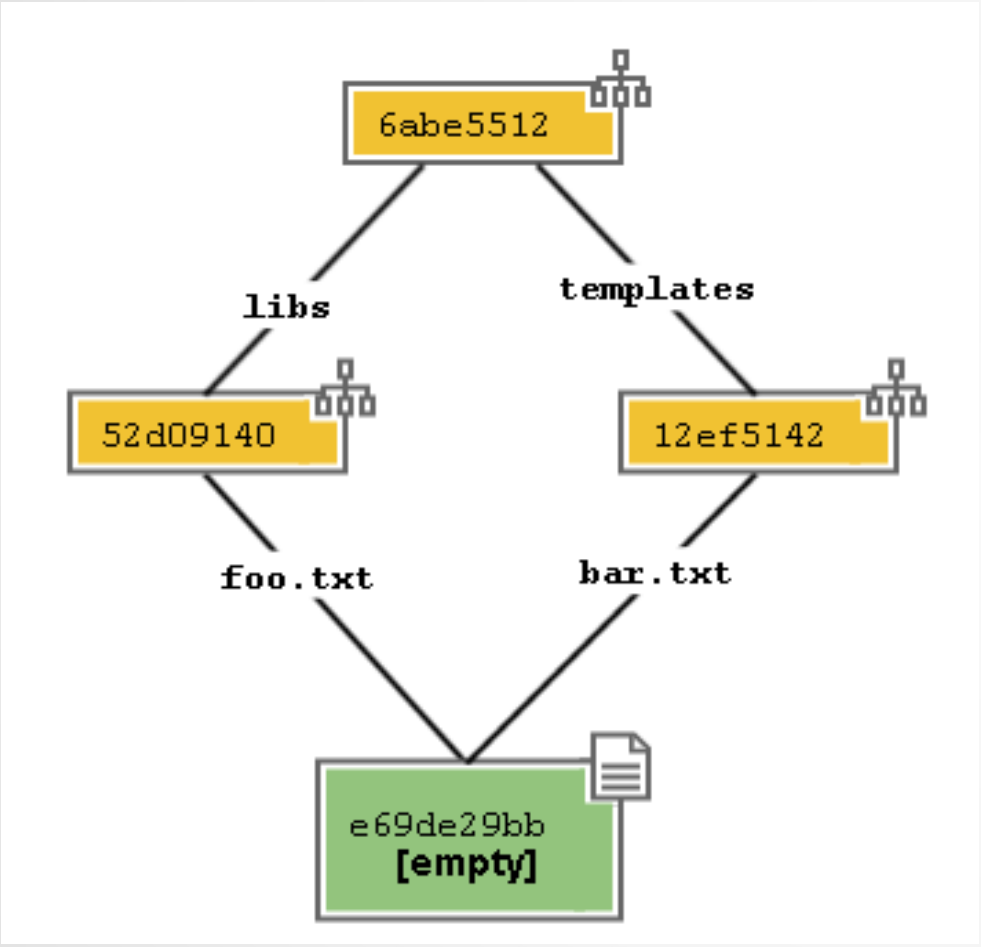
- Non c’è un server: il repository è locale. La gran parte delle operazioni è locale e non richiede l’accesso alla rete. Anche per questo git è incredibilmente veloce. (Riesce ad interagire con dei server remoti. Nonostante questo, resta sostanzialmente un sistema locale)
- Il progetto è indivisibile: 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)



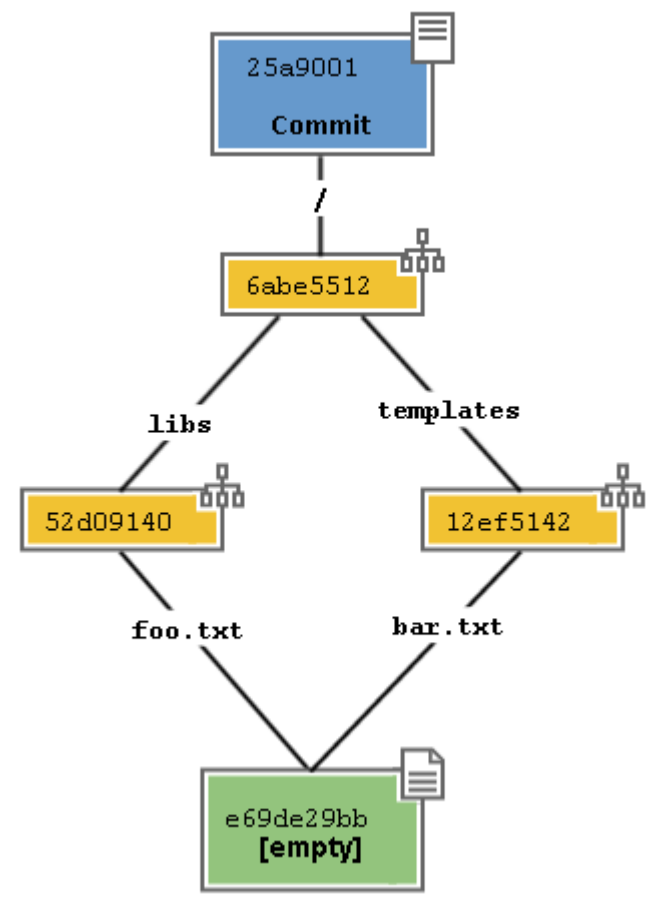
Root del progetto

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)

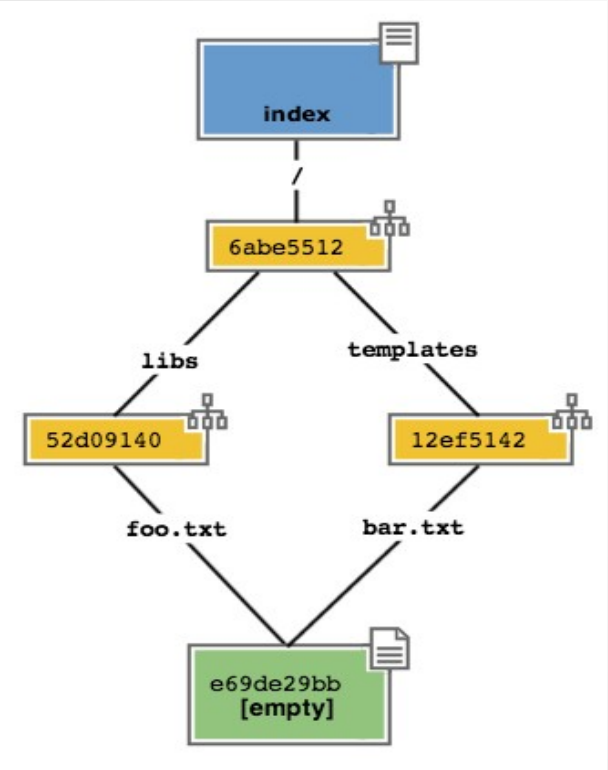
Staging area (Index) (1)

- 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.



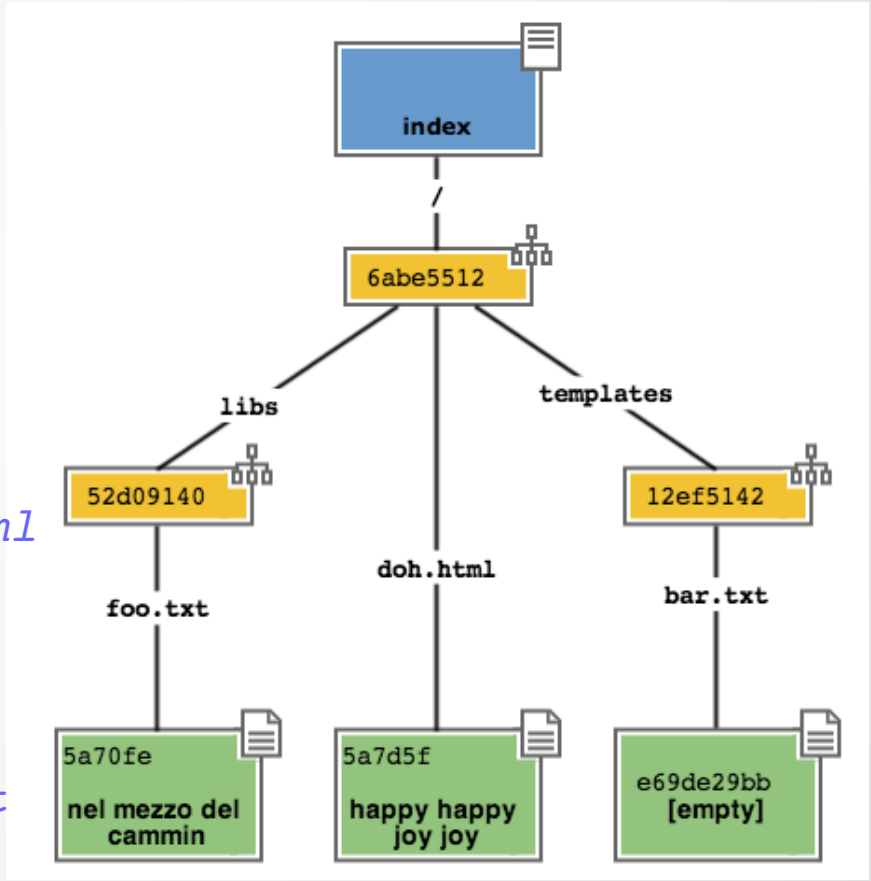
Aggiunta e modifica di un file nel index area

Comandi:

```
git add doh.html
```

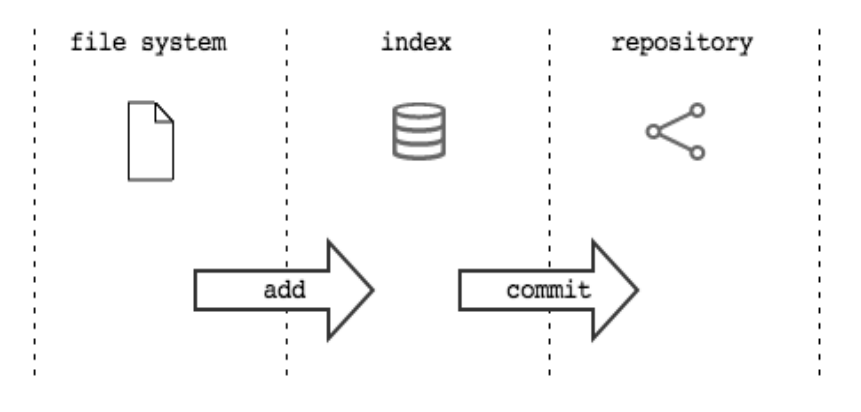
Modifico file di testo foo.txt

```
git add foo.txt
```



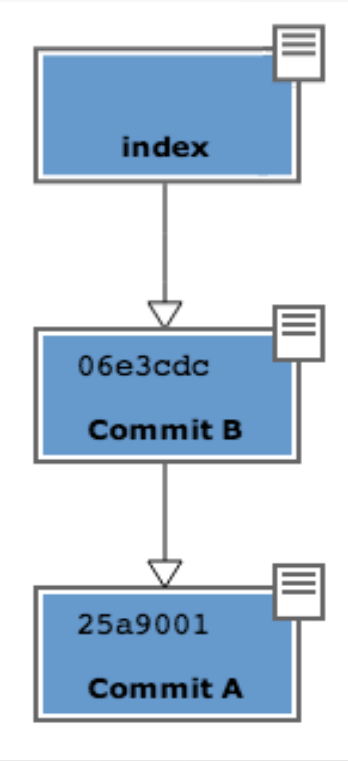
L'operazione add in git serve a salvare un file dentro l’index ed è un’operazione che va ripetuta ad ogni commit

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.

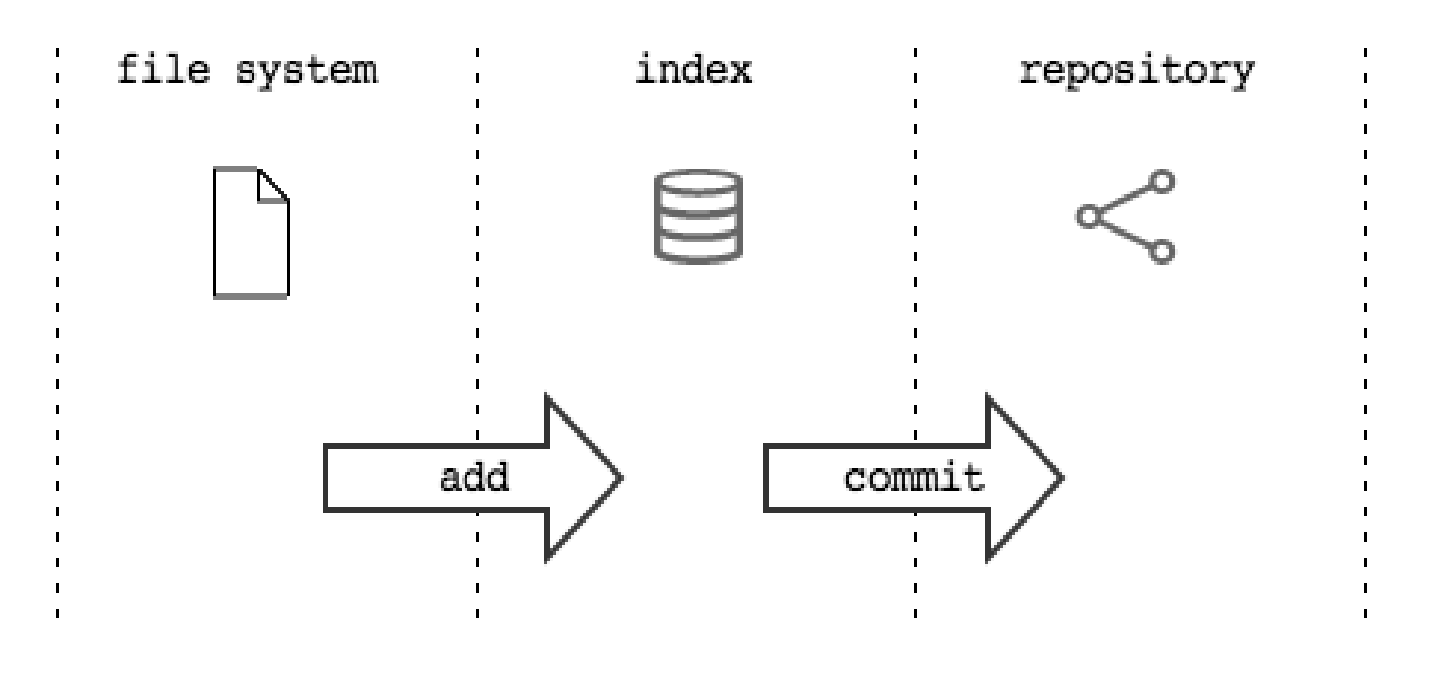
Ogni oggetto commit ha un puntatore al commit padre da cui deriva.
Per cui i commit sono esattamente la fotografia del file system in un dato momento.
Quindi posso ripercorrere la storia del progetto seguendo i puntatori ai commit padri.



Staging area (Index) (Ricapitolando)

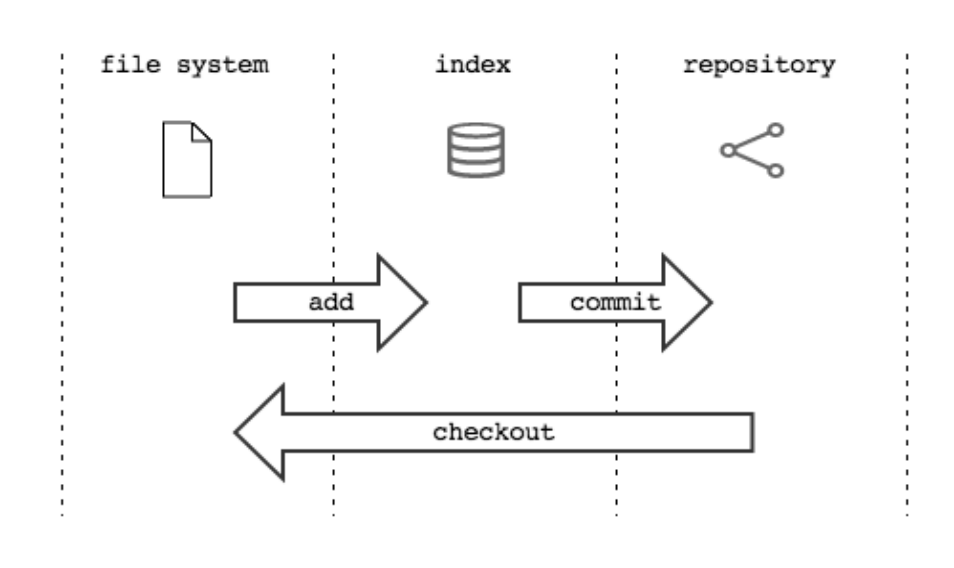
1) Git memorizza sempre i file nella loro interezza.
2) Il commit è uno dei tanti oggetti conservati dentro il database chiave/valore di git. È un contenitore di tanti puntatori ad altri oggetti del database: i tree, che rappresentano directory, che a loro volta puntano ad altri tree (sotto-directory) o a dei blob (il contenuto dei file).
3) Ogni oggetto commit ha un puntatore al commit padre da cui deriva
4) L'index è uno spazio di appoggio nel quale puoi costruire, a colpi di `git add`, il nuovo commit.
5) Con `git commit` registri l'attuale index facendolo diventare il nuovo commit.

Staging area (Index) (Ricapitolando)



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 li 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)

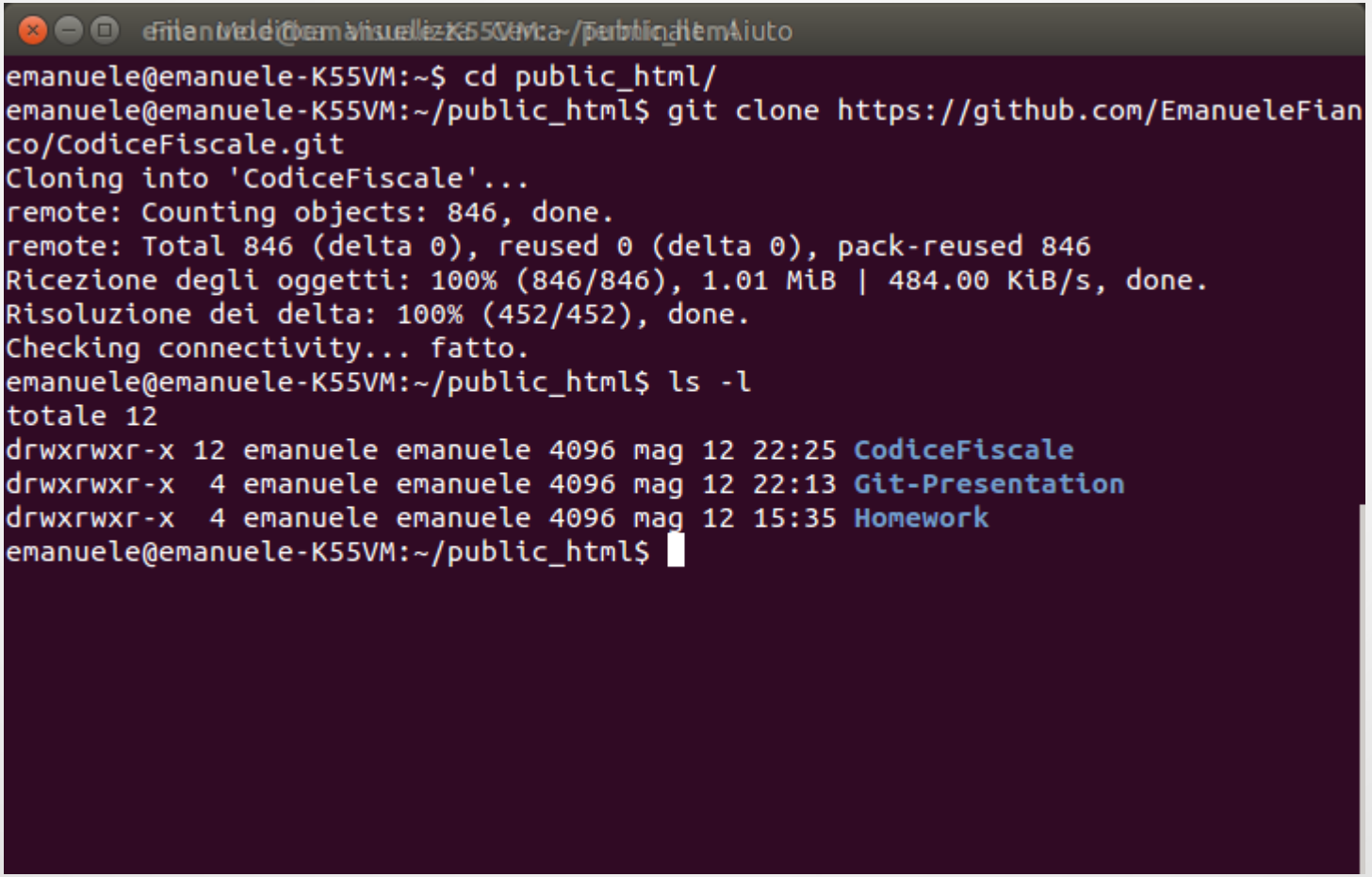
- *git remote add [NomeIdRemoto] “URL”*
- *git clone “URL”*
- *git fetch “NomeIdRemoto” [Branch]*
- *git pull “NomeIdRemoto” [Branch]*
- *git push “NomeIdRemoto” [Branch]*

git remote add [NomeIdRemoto] “URL”

- Questo comando aggiorna la lista dei repository remoti (possono essercene più di uno) aggiungendo la corrispondenza [NomeIdRemoto] → URL
- Il NomeIdRemoto è l'etichetta con cui identificare il repository remoto.
- L'URL è l'indirizzo del repository remoto che può essere anche un path (relativo o assoluto).

git clone "URL"

- Questo comando clona il repository remoto all'interno della cartella da cui viene lanciato il comando



git fetch “NomeIdRemoto” [Branch]

- Questo comando permette di allinearsi con il repository remoto indicato.
- Se non specificato il Branch voluto per default ci si allinea con tutti i Branch presenti in remoto.
- Tale comando cambia solamente lo stato del repository aggiornandoci lo stato del repository remoto.
- Il file system e i nostri rami locali non sono cambiati di una virgola.

git pull "NomeIdRemoto" [Branch] (1)

- Questo comando invece è l'insieme di due comandi che è possibile ovviamente eseguire separatamente:
  - *git fetch "NomeIdRemoto" [Branch]*
  - *git merge NomeIdRemoto[/Branch]*
- Si evince immediatamente che il comando *pull* rispetto a *fetch* è molto più invasivo (aggiorna il file system e i nostri rami locali facendo un'operazione di *merge*).
- Tale operazione di merge tuttavia è effettuata solo a patto che il tutto si possa risolvere con un fast-forward merge.

git pull "NomeIdRemoto" [Branch] (2)

- In conclusione quindi è possibile effettuare una *pull* solamente se in locale ci sono le condizioni adatte al più per un fast-forward merge.
- Se tali condizioni non sono rispettate ci sono 2 alternative:
  - Andare avanti con lo sviluppo in locale rimandando l'aggiornamento del repository remoto.
  - Effetture il *fetch* per poi fare un *merge* manualmente.

git push “NomeIdRemoto” [Branch] (1)

- con *git push “NomeIdRemoto” [Branch]* hai chiesto a git di spedire a “NomeIdRemoto” il ramo “Branch”
- per eseguire il comando git ha preso in considerazione il tuo ramo “Branch” ed ha ricavato l’elenco di tutti i commit raggiungibili da quel ramo (come al solito: sono tutti i commit che puoi trovare partendo da “Branch” e seguendo a ritroso nel tempo qualsiasi percorso tu possa percorrere)
- Attenzione a non confondere Branch con ramo!!

git push “NomeIdRemoto” [Branch] (2)

- git ha poi contattato il repository remoto per sapere quali di quei commit non fossero presenti remotamente.
- Ha creato un pacchetto con tutti i commit necessari, li ha inviati ed ha chiesto al repository remoto di aggiungerli al proprio database.
- il remote ha poi posizionato il proprio branch “Branch” perché puntasse esattamente lo stesso commit puntato sul tuo repository locale. Se il remote non aveva quel branch, lo crea.

Link utili

- http://get-git.readthedocs.org/it/latest/index.html
- http://git-scm.com/documentation/