Comandi in Shell

Andrea Savastano

December 31, 2024

Contents

1 Comandi bash

2 Commandi git

Introduzione

Questa sezione presenta un elenco di comandi git per gestire la propria repository in locale e per interagire con la repository in remoto sul server GitHub.

Si utilizza la shell `GitBash` perche è più simile a quella di `Linux`, in alternativa si può usare `PowerShell` o `cmd` di `Windows` che invece richiedono i comandi nativi di `Windows`.

E' possibile avere una repository git su un qualsiasi path (sul Desktop).
Una cartella si definisce repository git in locale al dispositivo quando è presente la sottocartella nascosta `.git`, che consente di usare i relativi comandi.
La repository in locale è diversa da quella che si ha su un server remoto ad esempio su GitHub, ed è possibile collegarle in modo che le modifiche (aggiunte o rimozioni di file) effettuate nella repository locale vengano caricate sul server, ossia sulla rispettiva repository remota.

2.1 Inizializzazione della repository

Se si desidera collegare la repository locale ad una remota, allora assicurarsi di aver creato prima la repository in remoto su GitHub (`https://github.com/new`) e di averne prelevato l'`"URL-remoto"`, che può essere HTTPS o SSH.

2.1.1 git clone

```
1 git clone "URL-remoto"
```

Clona la repository remota specificata dall'URL nel percorso in cui è stato eseguito il comando, creando una nuova cartella con lo stesso nome del remoto e aggiornata con gli stessi dati. Crea quindi una repository locale (`.git`), collegandola direttamente a quella remota.
In particolare si crea un branch `main` in locale che è collegato a quello remoto `remotes/origin/main`.
E' un comando molto più comodo e veloce per la creazione e il collegamento della repository, ma in alternativa si possono usare i comandi `git init` e `git remote`.

2.1.2 git init

```
1 git init
```

Inizializza la cartella in cui si esegue il comando come una nuova repository in locale (`.git`).
E' possibile lavorare solo in locale (tramite comandi come `git add`, `git commit`, ...), in quanto la repository locale non è stata collegata a nessuna repository in un server remoto.

2.1.3 git remote

```
1 git remote add origin "URL-remoto"
```

Collega la repository locale creata con `git init` con una remota già esistente.
In seguito a questo passaggio si ha che la repository in locale è collegata ad una nuova in remoto.

```
1 git remote set-url origin "URL-remoto"
```

Collega la repository locale ad un altro repository remoto, modificando l'URL del remoto esistente. Questo può servire quando si vuole collegare il repository locale a quello remoto con l'URL SSH, sostituendolo a quello HTTPS, o viceversa.
Oppure può servire per sostituire l'URL vecchio del repository remoto con uno nuovo, perchè è stato eventualmente cambiato.

```
git remote -v
```

Mostra l'elenco dei repository remoti associati al repository locale, insieme agli URL per `fetch` e `push`.

```
# Output di esempio:
origin  https://github.com/savaava/ShellCommands.git (fetch)
origin  https://github.com/savaava/ShellCommands.git (push)
```

Il nome dei repository remoti è di default `origin` e viene mostrato il repository remoto, tramite il suo URL, da cui si prelevano le commit (fetch) e quello su cui si caricano le commit (push).

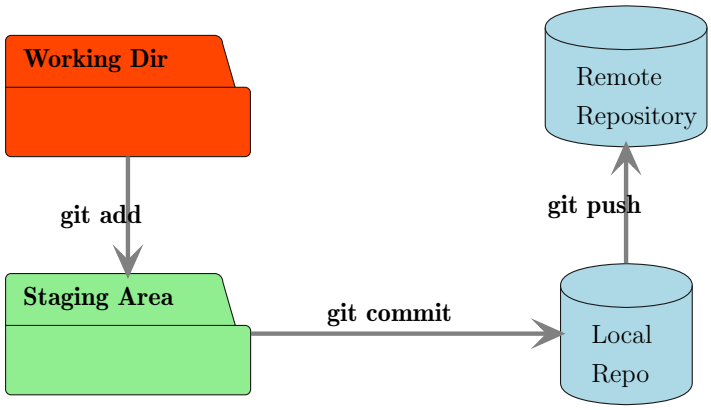
2.2 Aggiunta/caricamento

Quando si aggiorna la repository locale, quindi la working directory, aggiungendo/eliminando/modificando un file, è possibile effettuare una commit, la quale carica le modifiche in locale. Per aggiornare anche la repository remota a cui è collegata quella locale si deve effettuare una push della commit.

Pertanto, dopo aver aggiornato la repository locale, i passaggi da seguire sono 3:

1. `git add`
2. `git commit`
3. `git push`

I comandi si approfondiscono ai paragrafi successivi.



E' importante monitorare sempre lo stato della repository con `git status`, per tenere traccia dei cambiamenti in corso d'opera.

2.2.1 git status

```
git status
```

Mostra lo stato della repository locale, specificando i file che sono stati aggiunti, modificati o eliminati. Il comando in esame fornisce informazioni anche sui file che la repo sta monitorando:

- I file **Untracked** non sono ancora monitorati da Git (non aggiunti con `git add`), quindi sono i file aggiunti e non ancora caricati nella repo per la prima volta;
- I file **Tracked** sono già sotto il controllo di Git, quindi sono i file già caricati precedentemente nella repository, e possono essere **Unmodified**, **Modified** o **Deleted**.

I file Untracked, Modified e Deleted sono colorati in rosso nel terminale quando non sono stati ancora aggiunti alla staging area con `git add`. Una volta aggiunti saranno verdi.

```
# Output quando non sono state apportate modifiche:
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

In questo caso la repository in locale è sul branch `main` come suggerisce la riga 2, ed è collegato al branch remoto `origin/main` e sono entrambi aggiornati con gli stessi dati come suggerisce la riga 3. La riga 5 suggerisce che non vi sono modifiche aggiunte alla staging area per cui non c'è nulla da committare.

```
# Output quando e' stato creato, modificato ed eliminato un file:
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
 deleted:    file.txt
 modified:   file2.txt

Untracked files:
 fileProva.sh

no changes added to commit
```

2.2.2 git add

```
git add <file>...
git add .
```

Aggiunge alla staging area i file specificati, che sono stati creati, modificati o eliminati.
Quindi il comando aggiunge alla staging area le modifiche apportate alla repository rispetto alla versione precedente.
E' efficace utilizzare direttamente `git add .` perchè aggiunge velocemente alla staging area tutte le modifiche apportate alla working directory, quindi tutti i file aggiunti (`Untracked`), tutti i file modificati (`Modified`) e tutti i file eliminati (`Deleted`).

Rispetto all'esempio sopra proposto che prevede tre modifiche alla repo, eseguiamo `git add .` per aggiunge tali modifiche alla staging area e analizziamo lo stato della repo tramite `git status`:

```
# Output quando vengono aggiunte 3 modifiche:
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
 deleted:    file.txt
 modified:   file2.txt
 new file:   fileProva.sh
```

Si noti che ora le tre modifiche sono nella staging area e pronte per essere committate.

2.2.3 git commit

```
git commit -m "message"
```

Le modifiche presenti nella staging area vengono aggiunte nella repository git locale e non in remoto, raggruppandole in una commit, a cui si deve associare necessariamente un messaggio rappresentativo e si genera un codice identificativo (commit hash).
In questo modo la repo locale e quella remota non sono più aggiornate con gli stessi dati e si può effettuare una push della/delle commit in sospeso.

```
git commit -am "message"
```

Aggiungendo l'opzione `-a` è possibile saltare la staging area e aggiungere le modifiche direttamente alla repository locale come commit. Tuttavia non è in grado di aggiungere alla commit i file Untracked, i quali devono necessariamente passare per la staging area (`git add .`).

```
# Stato della repo quando e' stata effettuata una sola commit:
On branch main
Your branch is ahead of 'origin/main' by 1 commit.

nothing to commit, working tree clean
```

2.2.4 git push

```
git push
```

Carica le commit sulla repository remota a cui è collegata quella locale.
Di default la repository locale è sul branch `main` e `git push` carica le commit sul branch remoto `origin/main`.

2.2.5 git fetch

```
git fetch
```

Preleva le eventuali modifiche della repository remota e le carica in locale, senza aggiornare la working directory.

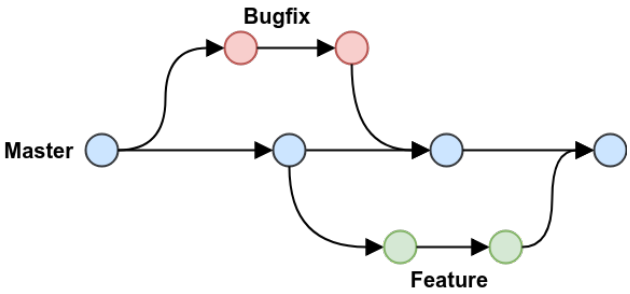
2.2.6 git pull

```
git pull
```

Preleva le eventuali modifiche della repository remota e le carica in locale, aggiornando la working directory, a differenza di `git fetch`.

2.3 Branch

I branch (rami) sono una delle potenzialità più rilevanti per Git perchè consentono ai contributori del progetto di lavorare in parallelo sullo stesso progetto, che si trova sempre in remoto (GitHub).
Branch differenti rappresentano linee di sviluppo indipendenti, quindi sono versioni diverse del progetto, con file differenti, perchè le modifiche si caricano su uno dei branch ausiliari e non più sul `main` o `Master`.
Sarà possibile eliminare il branch e perdere eventuali modifiche, oppure effettuare una merge (fusione) con quello `main` e ricongiungersi al percorso originale (come avviene nell'immagine).



Il branch rosso `Bugfix` e quello verde `Feature` sono branch ausiliari che si ricongiungono col `main`, dopo aver apportato le modifiche per cui sono stati creati.

Esattamente come per le repository, anche per i branch vi è una distinzione tra locale e remoto. Infatti:

- Un **branch locale** si trova nella repository Git locale al dispositivo;
- Un **branch remoto** si trova nella repository Git remota (GitHub).

E' molto utile collegarli perchè è possibile utilizzare i comandi `git pull` e `git push` senza dover specificare esplicitamente il branch remoto.
Collegare i branch significa configurare il branch locale affinché tracci il branch remoto.

2.3.1 git branch

```
1 git branch "nome-branch"
```

Crea un nuovo branch locale a cui si assegna il nome specificato.
Il branch si salva solo in locale e non in remoto, su cui è possibile caricarlo tramite i comandi che seguono.

```
1 git push --set-upstream origin <branch-locale>
```

Il `<branch-locale>` deve esistere, altrimenti restituisce un errore per cui non trova il branch specificato.
Crea un nuovo branch remoto con nome `remotes/origin/<branch-locale>` (se non esiste già), collegandolo al branch locale `<branch-locale>` specificato.
Se il branch remoto con nome `remotes/origin/<branch-locale>` esiste già allora non c'è la necessità di creare nessun branch remoto e il comando si limita a collegarlo al `<branch-locale>`.

```
1 git branch --set-upstream-to=origin/<branch-remoto> <branch-locale>
```

Entrambi i branch, `<branch-locale>` e `origin/<branch-remoto>`, devono esistere e in caso contrario, il comando restituirà un errore segnalando che il branch specificato non è stato trovato.
Il comando non crea nessun branch, nè in locale nè in remoto, bensì collega il branch remoto specificato (`origin/<branch-remoto>`) con quello locale specificato (`<branch-locale>`).
Questo comando è utile in alcuni scenari:

- Quando si ha già un branch remoto e uno locale, ma questi non sono ancora collegati;
- Quando si vuole cambiare il branch remoto tracciato da un branch locale.

```
git branch -a
```

Mostra tutti i branch attualmente esistenti sulla repository locale e remota, ed è utile per tenere traccia di tutti i branch. Questi sono distinguibili perchè:

- I branch remoti sono rossi e iniziano col prefisso `remotes/` seguito dal nome del remoto (`origin` di default) e il nome del branch, quindi `remotes/<nome-remoto>/<nome-branch>`;
- I branch locali sono rappresentati solo dal nome assegnatovi.

```
  # Output 1 di esempio:
* main
  remotes/origin/main
```

```
  # Output 2 di esempio:
* Bugfix
  main
  remotes/origin/Bugfix
  remotes/origin/main
```

In entrambi gli esempi esiste il branch locale `main` collegato al branch remoto `remotes/origin/main`, per cui se il branch corrente è il main le push caricano le commit sul corrispettivo branch remoto.
L'asterisco `*` indica qual è il branch attualmente attivo sulla repository locale.

```
git branch -vv
```

Mostra tutti i branch creati nella repository locale e i loro eventuali corrispettivi in remoto. Quindi indica se un branch locale è associato a un branch remoto, e inoltre mostra l'ultimo commit effettuato su ogni branch, specificandone il messaggio e il commit hash.
E' un altro comando di utilità insieme a `git branch -a`.
Il formato di output è: `branch-locale commit-hash [branch-remoto-associato] commit-message`.

```
  # Output 1 di esempio:
* Bugfix e65f5a9 [origin/Bugfix] update file.txt
  main   425f35d [origin/main] create file
```

```
  # Output 2 di esempio:
  Bugfix  e65f5a9 [origin/Bugfix] update file.txt
  branch2 425f35d create file
* main    425f35d [origin/main] create file
```

In questo secondo esempio il branch `branch2` non è associato a nessun branch remoto e quindi esiste solo in locale, ed è una diramazione del `main` perchè si nota che parte dalla stessa commit.

```
git branch -d <nome-branch>
```

Elimina il branch specificato dalla repository locale e non il suo eventuale corrispettivo remoto.

```
git push --delete origin <nome-branch>
```

Elimina il branch specificato dal remoto `origin` e non il branch in locale.

2.3.2 git checkout

```
1 git checkout <nome-branch>
```

Cambia il branch corrente con `<nome-branch>`, il quale deve necessariamente esistere.
Cambiare il branch attualmente in uso significa cambiare anche i dati della working directory con i file e il contenuto del branch selezionato, in quanto branch differenti rappresentano linee di sviluppo indipendenti (Figura a 2.3). Infatti branch differenti possono avere:

- file diversi, per cui un branch potrebbe contenere file che non esistono nel branch da cui ci si è spostati;
- stessi file (nome uguale) con contenuto differente.

2.3.3 git diff

```
1 git diff [options] <nome-branch>
2 git diff [options] <nome-branch1> <nome-branch2>
```

Mostra le differenze tra il branch corrente e il branch specificato nel comando (riga 1), mentre se vengono specificati entrambi i branch da confrontare (riga 2) il confronto viene effettuato tra questi, senza coinvolgere quello corrente.
E' possibile eseguire il comando senza alcun `[option]` per un risultato più generico. Tuttavia in base al criterio con cui si vogliono confrontare i due branch si possono usare 1 o più `[options]` specifiche; le più utili sono:

- `--stat` Mostra differenze di riepilogo tra i due branch (numeri di righe aggiunte/rimosse);
- `--diff-filter=<A|M|D|R|U|...>` Filtra le modifiche tra i due branch per tipo:
  `<A>` per aggiunte;
  `<M>` per modifiche;
  `<D>` per eliminazioni;
  `<R>` per rinomine;
  `<U>` Non unito (usato nei conflitti di merge).

Gli esempi che seguono si basano sulla seguente struttura:

```
1 # branch corrente: main
2 $ ls -1
3 fileContInv.txt
4 fileInvariato.txt
5 fileMain.txt
```

```
1 # branch corrente: bugfix
2 $ ls -1
3 fileBugFix.txt
4 fileContenutoInv.txt
5 fileInvariato.txt
6 fileMain.txt
```

- Il `main` conserva `fileContInv.txt`, mentre `bugfix` l'ha rinominato come `fileContenutoInv.txt`, senza cambiarne il contenuto;
- Entrambi i branch conservano lo stesso file `fileInvariato.txt` (stesso contenuto e nome);
- Il file `fileMain.txt` con lo stesso nome in entrambi i branch, ma con contenuto diverso;
- il branch `bugfix` aggiunge il file `fileBugFix.txt` che invece il main non possiede.

```
1 # Output di esempio 1:
2 $ git diff --stat main bugfix
3 fileBugFix.txt                          | 1 +
4 fileContInv.txt => fileContenutoInv.txt | 0
5 fileMain.txt                            | 2 +-
6 3 files changed, 2 insertions(+), 1 deletion(-)
```

```
# Output di esempio 2:
$ git diff --stat --diff-filter=A main bugfix
fileBugFix.txt | 1 +
1 file changed, 1 insertion(+)
```

```
# Output di esempio 3:
$ git diff --stat --diff-filter=M main bugfix
fileMain.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
# Output di esempio 4:
$ git diff --stat --diff-filter=R main bugfix
fileContInv.txt => fileContenutoInv.txt | 0
1 file changed, 0 insertions(+), 0 deletions(-)
```

2.3.4 git merge

```
git checkout main
git merge <nome-branch>
```

2.4 Stash

2.5 Altri

2.6 Commit e ripristino

2.7 Configurazione

3 Comandi ssh

Introduzione

Questa sezione spiega come gestire una connessione SSH tra il dispositivo client e quello server utilizzando OpenSSH (Open Secure Shell), che consente di eseguire operazioni remote su altri computer attraverso una rete in modo sicuro sulla porta 22.

Si utilizza `PowerShell` di `Windows` e `GitBash` eseguiti come amministratore, in alternativa è possibile utilizzare `cmd` sempre come amministratore.

3.1 Inizializzazione dei dispositivi

3.2 Apertura/Chiusura della connessione

3.3 Operazioni remote