1. Spiegare il ruolo di “origin”.

L’alias origin fa riferimento a una repository remota.

Ad esempio, nel comando **git pull origin develop**, chiediamo che il branch locale develop acquisisca tutti i contenuti del corrispettivo remoto. Questo risulta necessario ogni qual volta, ad esempio, lanciamo la shell di Git per continuare il nostro sviluppo, allora sarà fondamentale caricare sul branch di lavoro locale tutti i nuovi file (tracked) inseriti nel corrispettivo branch remoto durante il periodo di nostra inattività.

O allo stesso modo **git push –u origin develop,** chiede di pushare i file committati a livello locale nel branch remoto develop.

1. Comandi per creazione repository da riga di comando.

Ci sono due modi per creare una repository da riga di comando.

Se esiste già un progetto, allora faremo rifermento al seguente comando:

* **git clone [url-repository remota]**

questo permette non solo di creare una nuova repository, ma fa di pù, copia l’intero contenuto di una repository remota tracciata tramite il suo url (disponibile in genere su github).

Questo metodo è il più usato in quanto è molto più comune andare a lavorare su repository create in precedenza (fondamentalmente si collabora a progetti già esistenti).

Se invece siamo noi a creare un progetto ex novo, faremo riferimento alla seguente serie di comandi:

* **git init [nome-progetto]**
* **git remote add origin [url repo remota]**
* **git push -u origin master**

Il primo crea una repository locale all’interno di una directory già esistente sulla nostra macchina.

Ora, volendo tracciare il nostro progetto con Git, col secondo comando creiamo un riferimento remoto alla nostra repository, di modo che ogni utente possa collaborare a questo progetto tramite Git (e sarà possibile effettuare il clone di tale repository come nel primo metodo).

Il terzo comando infine crea il branch master all’interno della repository. (N.B. IL BRANCH MASTER è OBBLIGATORIO).

1. Eseguire una **gitpush force** e spiegare le conseguenze di tale azione.

**Git push origin develop --force**

Fondamentalmente questo comando aggira le limitazioni del semplice **push**, che permette di passare i commit nel branch di riferimento remoto solo se questi hanno come “antenato” il branch di destinazione, in questo caso develop (ad esempio dopo un **pull**). Il **push --force** quindi permette di pushare in remoto anche una repository creata ex novo in locale, però porta con se dei rischi, perché fondamentalmente sovrascrive la storia del branch remoto, con la perdita delle commit pushate precedentemente.

1. Ogni commit ha un id univoco. Documentatevi sul come tale id viene assegnato. (non è necessario scendere troppo nel dettaglio).

Ogni commit che effettuiamo è legata ad un ID, prodotto tramite un algoritmo, denominato  **SHA-1** (di cui esistono 4 varianti ) che genera un codice di 40 cifre in base esadecimale.

Questo codice contiene diverse info relative alla commit, tra le più importanti:

* Contenuto, (senza le diff)
* La data della commit
* User e email di chi effettua la commit
* Il messaggio della commit
* L’id della commit precedente

Questo ID risulta valido per vari scopi, ma in particolare:

* Verificare una eventuale manomissione della commit (argomento molto ampio)
* Permette di confrontare due commit senza effettuare il diff, ma solo tramite il semplice confronto degli ID (rendendo i protocolli di rete di Git molto efficienti )

(N.B. commit identiche effettuate da macchine differenti hanno lo stesso ID)

1. Spiegare differenza fra una revert e una reset, con relativi esempi .

Con il comando **revert** si mette in coda alle varie commit una nuova (commit), che fa riferimento ad una esistente, con la particolarità che il file di tale commit non presenterà più le modifiche effettuate durante lo sviluppo. Quindi nel push la commit di partenza (col file modificato) verrà pushata, ma successivamente sovrascritta dalla commit “revertata”. Inoltre questo comando lascia invariata la cronologia delle modifiche sul file in oggetto a livello di working directory.

**Git revert f3abe64fc121b75f3f0566c73f2f1a4e8fffd68e**

Con il comando **reset** elimina del tutto una determinata commit, la elimina anche dalla stage area, lasciando inalterata la working directory.

Senza specificare una specifica commit, si riferisce all’ultima commit effettuata a livello cronologico.

**Git reset --hard** invece cancella tutte le commit e pulisce del tutto la stage area.

**Git reset HEAD [commit]** effettua l’unstage di un determinato file precedentemente addato.

1. L’integrazione due branch può essere effettuata tramite due comandi: **marge e rebase.**

La prima non tiene conto della storia delle varie commit la secondo invece fa il confronto commit-commit valutando le diff. Entrambe generano un branch di appoggio dove risolvere i vari conflitti, per poi effettuare nuovamente un add . (di tutti i file con conflitti risolti), un nuovo commit totale dal dal branch di appoggio a quello di lavoro, e da qui possiamo finalmente effettuare un push senza avere più conflitti, a meno che qualche altro utente nel frattempo non ha pushato altre modifiche che possano creare nuovi conflitti.