

GFT ■

Sistemi di versionamento: Git



“Computer programs are the most complex things that humans make.”

Douglas Crockford, JavaScript: The Good Parts

Obiettivo

Il corso mira a formare programmatori capaci di lavorare a progetti di sviluppo di applicazioni complesse lavorando in team, condividendo il codice tramite l'uso di sistemi di controllo versione distribuiti, nello specifico GIT.



Utili strumenti

Software da scaricare:

- [GIT](#) download git

Opzionale:

- [Tortoise.git](#) un famoso client git
- [Learngitbranching](#) esercizi interattivi su git, consigliati da fare dopo la fine del corso



Version Control System

I sistemi di versionamento, chiamati anche sistemi di controllo versione, sono sistemi che si occupano della gestione di versioni multiple del codice e sono divenuti indispensabili per la realizzazione di progetti di sviluppo software.



Ok, ma nel concreto
quali problemi tentano
di risolvere?

Molteplici ambienti

Nello sviluppo software vengono utilizzati molteplici ambienti che utilizzano diverse versioni del codice:

- Produzione
- Test, UAT
- Locale

Tutti questi ambienti hanno versioni diverse, ed è complesso tracciare ogni modifica eseguita che le rende differenti.

Effettuando il passaggio di codice da un ambiente all'altro si rischia molto facilmente di perdersi pezzi, portare modifiche non volute e introdurre molti bug difficili da analizzare e tracciare.

Molteplici sviluppatori

Solitamente i team sono composti da più sviluppatori che lavorano in parallelo a sviluppi diversi sulla stessa versione iniziale del codice.

Una volta finiti gli sviluppi individuali, come si fa ad unirli per mandarli nell'ambiente di test?
Bisognerebbe tenere traccia manualmente di ogni singola riga di codice modificata, e poi trovarsi tutti assieme per unire le modifiche

Git

Git è il sistema di versionamento open source più utilizzato in assoluto.

Esso permette di:

- Tracciare ogni modifica ai file.
- Creare diverse versioni dello stesso progetto.
- Risolvere i conflitti tra modifiche simultanee ai files.
- Conservare l'intera cronologia di sviluppo e condividerla tra tutti gli sviluppatori.
- Ottenere istantaneamente una qualsiasi versione dei file di progetto.
- Invertire una qualsiasi modifica.



git

Repository

La repository centrale è uno spazio sul web che contiene l'intero progetto e l'intera cronologia delle modifiche. Viene messo a disposizione degli utenti abilitati, che potranno accedere con le proprie credenziali, per effettuare operazioni di download e di upload.

Esempi: Gitlab, GitHub, Bitbucket, CodeCommit.

Ogni sviluppatore può configurare sul proprio PC una repository locale (una cartella con git inizializzato) dedicata al progetto. Questa dovrà essere collegata alla repository remota, tramite il suo url, per permettere lo scambio di informazioni.

La repository locale è quasi completamente autonoma e può «sincronizzarsi» manualmente con quella remota, caricando o scaricando dati attraverso specifici comandi lanciati dallo sviluppatore.

```
graph TD; A[Repository centrale] --- B[Repository Locale 1]; A --- C[Repository Locale 2]; A --- D[Repository Locale n];
```

Repository centrale

Repository
Locale 1

Repository
Locale 2

Repository
Locale n

Repository

In locale, per creare una repository git, eseguire i seguenti passaggi:

- Aprire un linea di comando (per Windows consiglio la Git Bash installata assieme a git stesso)
- Posizionarsi sulla cartella sulla quale si vuole inizializzare la repository git
- Eseguire il seguente comando:
> git init

Verrà creata una cartella nascosta .git che conterrà tutti i dati riguardanti il versionamento del contenuto della cartella prescelta.

Remote - origin

Creata una repository locale, questa può essere collegata a una repository remota, in modo da poter comunicare con essa per scaricare e inviare dati.

- Il comando per creare il collegamento è il seguente:
> git remote add origin *remote_repository_url*
- “origin” è il nome che si dà, per convenzione, alla repository remota, ma esso può essere un nome diverso.

A una repository locale possono essere collegate più repository remote, in tal caso le altre repository devono avere un nome diverso da origin.

- Per visualizzare le informazioni sulla connessione remota impostata, eseguire:
> git remote -v

il risultato sarà:

origin remote_repository_url (fetch) ← rappresenta l'url per fare un download

origin remote_repository_url (push) ← rappresenta l'url per fare l'upload

Commit

Il commit rappresenta il salvataggio dello stato dei file, nuovi o modificati o cancellati, in un preciso istante della repository.

I commit risolvono il problema di tracciare ogni modifica ai file e sono i mattoni fondamentali della cronologia del progetto.

Un commit è costituito da:

- Un **hash**, ovvero un codice che lo identifica univocamente
- Il **diff**, ovvero tutte le modifiche ai file contenute in quel commit
- Un **messaggio** inserito dallo sviluppatore nel momento in cui effettua il commit
- Timestamp e autore del commit

Commit - comandi

Il comando per effettuare un commit è:

> **git commit -m "messaggio di commit"**

Il messaggio di commit deve essere descrittivo delle modifiche effettuate (descrizione del bug risolto o della funzionalità sviluppata), in modo che si possa avere un'idea del contenuto al volo semplicemente leggendolo.

```
commit dbd78d240fdacf750a784291dd494e581016a20b
Author: a38v <daryl.marra@gft.com>
Date: Tue Jan 25 10:43:34 2022 +0100

    corrected swagger examples

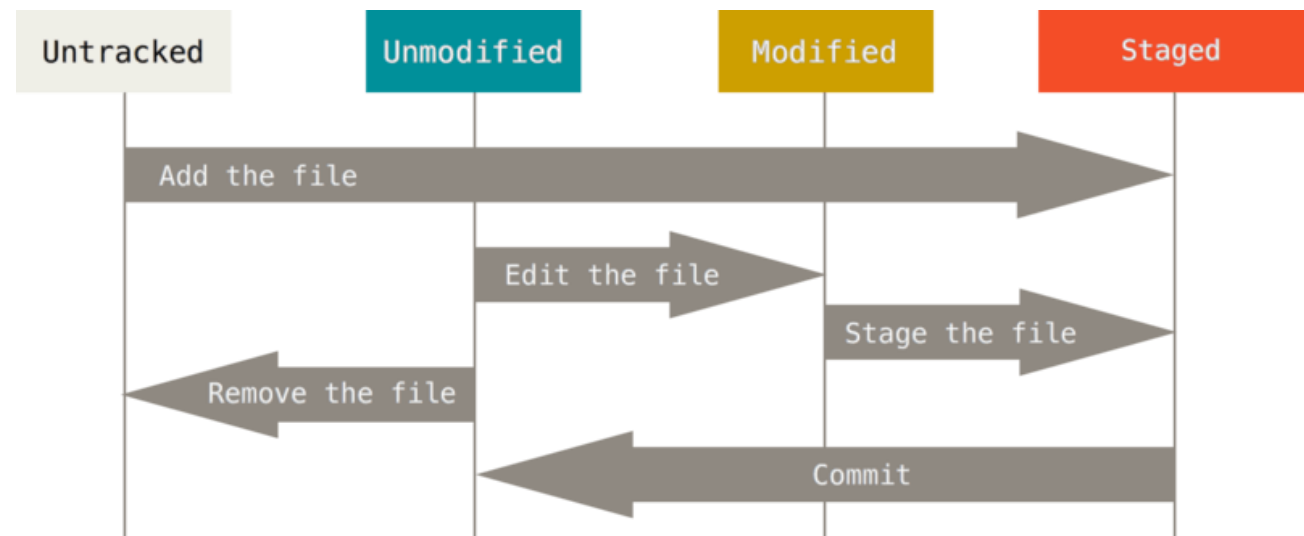
diff --git a/route/route.js b/route/route.js
index 2372bd6..a84a564 100644
--- a/route/route.js
+++ b/route/route.js
@@ -9,7 +9,7 @@ exports = module.exports = [
    effectiveDate: {
      type: 'string', //string integer number boolean array object
      description: 'effective date to start the policy',
-     example: '2022-01-20T00:00:00.00056Z'
+     example: '2022-01-20T00:00:00'
    },
    planDuration: {
      type: 'integer',
```

Stato dei files

I files presenti all'interno del progetto git possono essere suddivisi in 3 stati:

1. **MODIFIED:** files modificati la cui modifica non è ancora stata “committata” all'interno della repository
2. **STAGED:** i files modificati possono essere marcati per essere inseriti all'interno del prossimo snapshot che verrà creato col commit, quando succede i file passano in stato staged
3. **COMMITTED:** files salvati all'interno della repository attraverso lo snapshot eseguito dal commit. Questi files sono ora considerati come non modificati (unmodified) in quanto la loro modifica è stata registrata nella cronologia di progetto

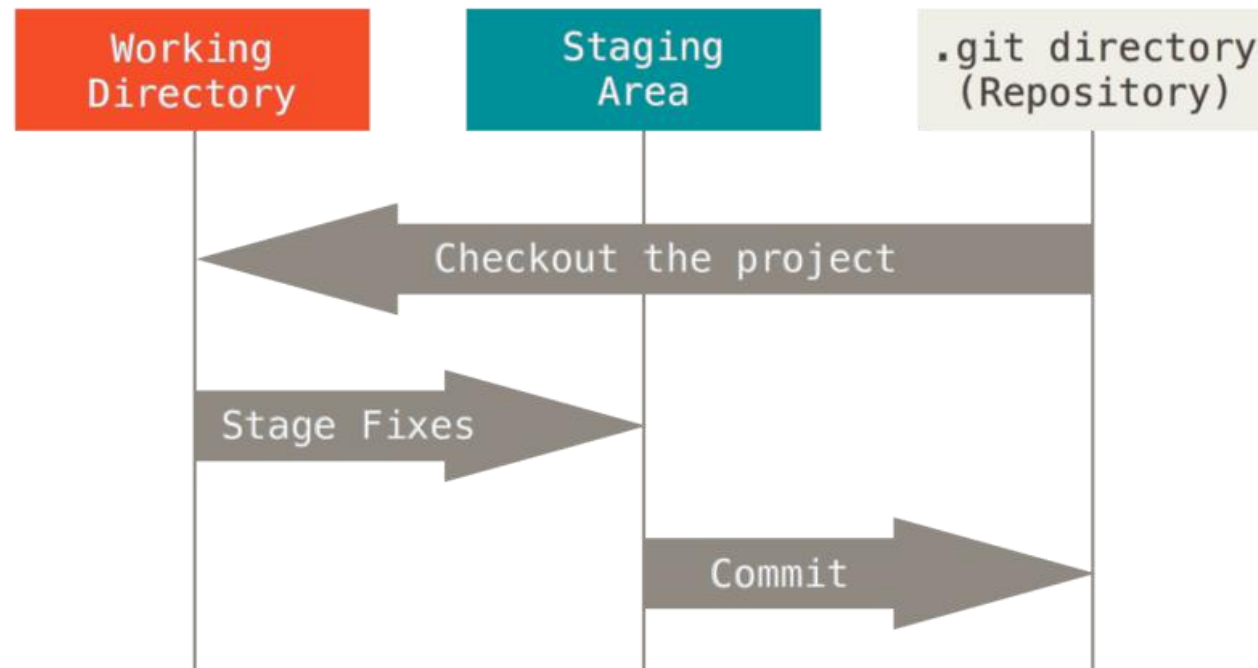
A questi stati si aggiunge un quarto stato chiamato **UNTRACKED:** i files in questo stato non sono ancora aggiunti al progetto (ovvero non sono tracciati da git), questo è lo stato di default in dei nuovi files creati all'interno della cartella.



Stato dei files

A questi primi tre stati corrispondono le tre sezioni di un progetto git:

1. **Working Tree** (chiamato anche **Working Directory** o **area di lavoro**) ove risiedono i files in stato modified assieme alla versione corrente dei files non modificati
2. **Staging Area** ove risiedono i file marcati come staged
3. **Cartella .git (la Repository locale)** ove risiedono tutti i files committati in ogni versione



Stato dei files - comandi

Per scoprire lo stato dei files basta semplicemente lanciare il comando **> git status** il quale mostrerà tutti i files staged, modified e untracked.

Per marcare in stato staged un file modified o untracked si lancia il comando :

> git add path/to/nome_file.ext

Oppure, per aggiungere tutti i file modified e untracked in una sola volta:

> git add .

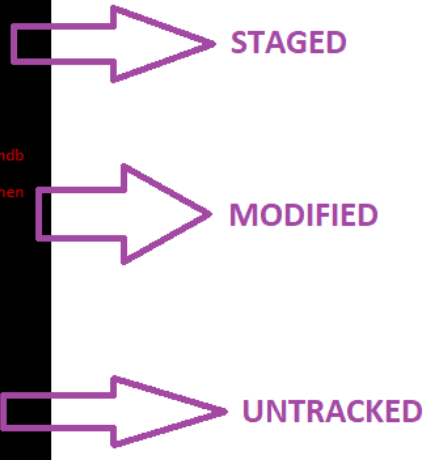
Invece per committare tutti files staged si usa il comando **> git commit -m “messaggio di commit”** visto in precedenza.

```
a38v@ITPC009937 MINGW64 /c/Guidewire/AvAsp/policycenter (release/r-6.3)
$ git status
Refresh index: 100% (33858/33858), done.
On branch release/r-6.3
Your branch is up to date with 'origin/release/r-6.3'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   local_portals_auth.patch

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   modules/configuration/config/database-config.xml
    modified:   modules/configuration/config/runtimeproperties/Integrationdb
RuntimeProperties.xml
    modified:   modules/configuration/gsrc/edge/oauth/authplugin/OAuthAuthen
ticationSource.gs
    modified:   modules/configuration/gsrc/gw/auth/AuthServicePlugin.gs
    modified:   studio/plugins/availables.xml
    modified:   studio/plugins/availables.xml.etag
    modified:   studio/plugins/ij-studio/lib/studio-branding.jar
    modified:   webapps/pc/resources/js/gen/all.js

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    AGRU-1174.patch
    Commenti.patch
    HFM.patch
    PolicyCenter.bat
    Problema_date.patch
```



Branch

Un branch rappresenta un insieme, o una successione, di commit.

Se un singolo commit rappresenta le modifiche fatte a determinati file, i branch sono allo stesso tempo sia la fotografia dello stato del progetto in quel preciso istante che la storia, o timeline, di esso.

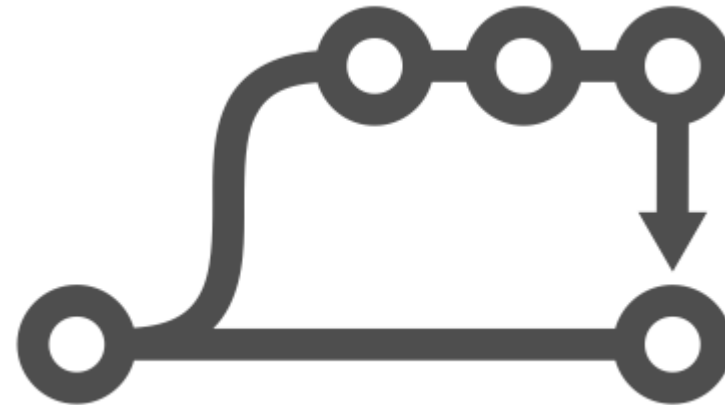
Più precisamente:

- Il branch è un puntatore a un commit. Facendo nuovi commit, il puntatore si sposterà automaticamente sull'ultimo commit.
- Al branch viene dato un nome univoco, per esempio "master", "main", "dev", "release", ecc.
- In un progetto posso coesistere tantissimi branch e molti branch possono puntare allo stesso commit.
- I branch permettono agli sviluppatori di lavorare contemporaneamente sul codice

Branch

- I branch permettono di lavorare contemporaneamente su due o più versioni diverse del progetto (controllo delle versioni).
- Un branch può essere unito (mergiato) a un altro branch portando così su quest'ultimo tutte le modifiche fatte sul branch mergiato.

I branch permettono quindi di risolvere il problema di ambienti differenti e anche il problema di molteplici sviluppatori che lavorano in parallelo, assegnando branch diversi a ciascun ambiente e/o sviluppatore e permettendo di sincronizzarli in maniera controllata e semplice



Branch - comandi

- Per creare un nuovo branch (il nuovo branch creato punterà allo stesso commit al quale siamo posizionati):
> **git branch nome_branch**
- Per eliminare un branch:
> **git branch -D nome_branch_da_eliminare**
- Per visualizzare la lista dei branch, anche i branch "origin/", eseguire:
> **git branch --all**
il branch sul quale siamo posizionati sarà evidenziato in verde
- Per verificare i commit all'interno del branch attuale:
> **git log** (si può utilizzare l'opzione **--online** per una versione più compatta)

Checkout

Il checkout è una funzionalità di git che permette di spostarsi in un determinato branch o in un determinato commit, portando tutti i files (il Working Tree) allo stato del branch o del commit in cui ci si è spostati.

- Possiamo spostarci indietro nei commit per analizzare il codice vecchio, “viaggiando nel tempo”.
- Possiamo spostarci su un diverso branch, per esempio un branch che contiene sviluppi fatti da un collega che ci servono oppure il branch di produzione per risolvere un bug.

Il branch corrente dell’area di lavoro è chiamato **HEAD**, è un puntatore all’ultimo branch o commit di cui è stato effettuato il checkout.

Checkout - comandi

- Per effettuare il checkout e posizionarci su un altro branch:
> **git checkout *nome_branch***
- Per fare il checkout di un determinato commit:
> **git checkout *hash_del_commit***

per risalire all'hash del commit, possiamo recuperarlo eseguendo il comando:
> **git log --oneline**

È possibile eseguire il checkout solo di uno o più file, senza doverlo eseguire su tutta la repository, in questo modo si ripristina il file allo stato che ha in quel determinato commit o branch:

> **git checkout *nome_branch* -- *cartella1/cartella2/file1.txt cartella3/cartella4/file2.txt***
> **git checkout *hash_del_commit* -- *cartella3/cartella4/file3.css cartella3/cartella4/file3.css***

Operazioni remote

Esistono 2 tipi di branch:

- **Branch remoti:** sono branch nascosti e non direttamente modificabili. Sono una copia esatta dei branch che si trovano nella repository remota. Essi sono contraddistinti dal prefisso “origin/” seguito dallo stesso nome del branch che si trova in remoto.
- **Branch locali:** sono i branch creati localmente dallo sviluppatore sui quali esso può committare e lavorare liberamente. Lo sviluppatore può anche creare copie locali e quindi modificabili di branch remoti, ma essi non saranno sincronizzati automaticamente.

Fetch

È un'operazione di download dalla repository remota.

- Tutti i nuovi branch presenti nella repository remota vengono scaricati nella nostra repository locale con il nome `origin/nome_branch`.
- Tutti i branch remoti ("origin/") presenti sul nostro locale vengono aggiornati con le ultime modifiche scaricate.
- È un operazione innocua, non comporta modifiche alla nostra area di lavoro. Si può eseguire in qualunque momento senza preoccupazioni.

> **git fetch**

Push

Rappresenta l'upload sulla repository remota del branch.

- Se si pusha un branch già presente nella repository remota, questo verrà aggiornato con gli ultimi commit effettuati in locale.
- Se si pusha un branch nuovo, esso verrà aggiunto alla repository remota con ogni suo commit.
- Il comando per eseguire un push è il seguente:
> git push -u origin *nome_branch*
il flag **-u** collegherà il branch locale a quello remoto, in modo che push successivi di quel branch possono essere eseguiti semplicemente col comando: **> git push**
- Terminato il push, il branch pushato sarà disponibile nella repository remota e potrà essere scaricato dagli altri sviluppatori.

Merge

Serve per aggiungere a un branch il contenuto di un altro branch.

- Tipicamente esiste un branch principale sul quale vengono riportati tutti gli sviluppi.
- Gli sviluppatori lavorano separatamente, sui propri branch.
- Questi branch vengono poi mergiati nel branch principale, così da includerci tutti i nuovi sviluppi fatti dai diversi sviluppatori.
- Il comando per eseguire il merge è il seguente:
> git merge *nome_branch_da_mergiare*

Assumendo che il branch corrente sia *dev*, questo comando andrà a mergiare *nome_branch_da_mergiare* nel branch *dev*, riportando i commit presenti in *nome_branch_da_mergiare* all'interno di *dev*.

- Questo comando viene utilizzato raramente in locale ma molto frequentemente in remoto attraverso le merge/pull request.

Pull

Comando usato molto frequentemente per aggiornare il branch corrente.

- Questo comando è composto dall'unione di due comandi visti precedentemente: fetch + merge.
- Inizialmente scarica dati di un branch presente nella repository remota e, automaticamente, effettua il merge di quel branch sul branch locale corrente.
- Serve per allineare il proprio branch alle ultime modifiche presenti nel branch remoto.
- Oppure per scaricare (pullare) le modifiche fatte nei commit di un altro branch e mergiarli sul proprio branch corrente.
- **> git pull origin *nome_branch_remoto***
- La differenza fondamentale con merge è che mentre nel caso di merge sia il branch di destinazione che il branch sorgente sono locali, invece con pull il branch sorgente è remoto (mentre quello di destinazione rimane sempre locale).

Conflitti

Un'operazione di merge o pull può fallire perché un file modificato da noi potrebbe essere stato modificato da un altro sviluppatore.

- Git ci chiederà di analizzare le modifiche fatte da noi e quelle fatte dall'altro sviluppatore per ogni singolo file che presenta dei conflitti.
- Dobbiamo decidere quali modifiche mantenere e quali scartare, oppure modificare ulteriormente i files.
- Una volta deciso, occorre correggere i files in conflitto, poi si prepara (con **git add .**) e si effettua un nuovo **commit** di tutti i files.
- Se invece si decide di non procedere al merge e si vuole ripristinare il branch allo stato precedente al tentativo di merge basta lanciare il comando **git merge --abort** durante la fase di merge con conflitti.

Merge Request / Pull Request

È un'operazione che si effettua sull'applicazione messa a disposizione per gestire la repository remota, per esempio Gitlab, GitHub, Bitbucket, CodeCommit. Viene utilizzata per richiedere di mergiare un branch pushato da uno sviluppatore in uno dei branch principali del progetto.

- Il branch, prima di essere mergiato, viene spesso sottoposto a una code review.
- Il reviewer avrà a disposizione un'interfaccia che mostra le modifiche effettuate e le differenze rispetto al branch sul quale si vogliono mergiare le modifiche.
- Se il codice presenta qualche imperfezione, il reviewer chiederà una correzione allo sviluppatore, che dovrà correggere il codice sul suo locale, effettuare un commit con le ultime correzioni e infine pushare nuovamente il branch.
- Subito dopo il push, la merge request verrà aggiornata con le ultime correzioni e il reviewer potrà mergiare il branch dello sviluppatore nel branch principale.

Operazioni di merge

Operazione Branch	Merge	Pull	MR/PR
Sorgente	Locale	Remoto	Remoto
Destinazione	Locale	Locale	Remoto

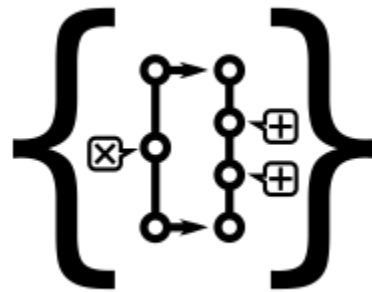
.gitignore

È un file che viene inserito nella cartella base di un progetto. Contiene la lista tutti i file che devono essere ignorati, cioè quei file che non vogliamo includere quando effettuiamo un salvataggio dello stato del progetto, quindi non andranno mai nella Staging Area.

Aggiungendo una nuova riga al .gitignore è possibile ignorare:

- uno specifico file, specificandone il path e l'estensione (esempio: nome_cartella/file.txt)
- il contenuto di un'intera cartella, specificandone il path (esempio: nome_cartella/)
- tutti i files di una certa estensione (esempio: *.xlsx)
- e molto altro ancora...

ATTENZIONE: gitignore non può ignorare files già committati in precedenza, i files che non si vuole condividere su git non vanno mai committati neanche una volta.



Operazioni utili

Cherry-pick

Da un branch, è possibile prendere un singolo commit e spostarlo su un altro branch, riportando su quest'ultimo solo le modifiche fatte ai file presenti nel commit.

- È simile al merge, ma invece che riportare un set di commit sul branch desiderato, si porta uno e un solo commit.
- Il commit cherry-pickato viene inserito in cima al branch e il puntatore del branch viene spostato su quest'ultimo commit.
- Il comando per eseguire il cherry-pick è il seguente:
> git cherry-pick *hash_del_commit*
- È possibile incorrere in conflitti durante questa operazione. In tal caso basta risolvere il conflitto, come spiegato nelle slide precedenti, e procedere a effettuare un commit.

Reset

Il comando reset serve a riportare il branch corrente allo stesso stato di uno dei branch che abbiamo in locale.

- Se il nostro branch presenta dei problemi, è possibile resettarlo a uno dei branch più stabili per ripristinare la situazione.
- Le copie dei branch remoti che abbiamo in locale (i branch origin/) sono un'ottima scelta per resettare i branch correnti. Prima di effettuare i reset a questi branch, è utile eseguire un fetch per aggiornare tali branch con le ultime modifiche dalla repository remota.
- Il comando per eseguire il reset è il seguente:
> git reset --hard origin/nome_branch

Altri comandi utili

- **git revert *hash_del_commit*** crea un nuovo commit contenente le modifiche opposte all'hash del commit specificato, di fatto annullandole
- **git clone *remote_repository_url*** in un solo comando inizializza la cartella git, la collega all'origin remota, scarica i branch e fa un checkout del branch di default
- **git checkout -b *nome_branch*** (notare il flag **-b**) crea il branch col nome prescelto a partire dal branch corrente e ne fa direttamente il checkout, corrisponde all'unione in un solo comando di **git branch *nome_branch* + git checkout *nome_branch***
- **git branch -vv** mostra tutti i branch locali, indicando anche il branch remoto collegato e hash e nome del commit a cui punta

Workflow di sviluppo

In un progetto che utilizza git, solitamente quando si lavora ad uno sviluppo vengono effettuate le seguenti operazioni:

1. **Checkout** del branch di riferimento da cui iniziare lo sviluppo
2. **Pull** del branch di riferimento per aggiornare la repository locale scaricando le ultime modifiche dalla repository remota
3. Creazione e checkout di un nuovo branch locale di sviluppo
4. Sviluppo: modifica file esistenti, creazione di nuovi file e cancellazione di file inutilizzati
5. Preparazione: si inseriscono i file da committare nella staging area col comando **add**
6. **Commit**: si effettua un salvataggio dello stato del progetto in quel preciso istante inserendo un messaggio con una appropriata descrizione degli sviluppi presenti in quel commit
7. Eventualmente ripetere i punti 4-6 per quanto risulta necessario al completamente dello sviluppo del branch

...

Workflow di sviluppo

8. **Push** del branch locale di sviluppo
9. **Pull** del branch di riferimento per aggiornare la repository locale scaricando le ultime modifiche dalla repository remota
10. Risoluzione di eventuali conflitti e **commit** della risoluzione
11. Secondo **Push** del branch locale di sviluppo
12. Creazione della Merge Request/ Pull Request a cui fare la code review

GFT ■

Sistemi di versionamento: Git



“Computer programs are the most complex things that humans make.”

Douglas Crockford, *JavaScript: The Good Parts*