

# Sistemi di versionamento: Git

*Daril Marra*



## Sommario

VCS: cosa sono e perché.....	3
Perché utilizzare un sistema di versionamento.....	3
Molteplici ambienti .....	3
Molteplici sviluppatori.....	3
Git .....	4
Repository.....	5
Commit .....	6
Stato dei files .....	7
Branch.....	9
Checkout.....	10
Operazioni remote.....	11
Fetch .....	11
Push .....	11
Merge .....	11
Pull .....	12
Conflitti .....	12
Merge Request/ Pull Request.....	12
.gitignore.....	14
Altri comandi utili .....	15
Workflow di sviluppo.....	16

## VCS: cosa sono e perché

I sistemi di versionamento (in inglese VCS: version control system), chiamati anche sistemi di controllo versione, sono sistemi software che si occupano della gestione di versioni multiple del codice di un progetto di sviluppo software e sono divenuti indispensabili per la realizzazione di essi.

### Perché utilizzare un sistema di versionamento

Per capire bene cosa sono i sistemi di versionamento bisogna comprendere quali problematiche relative allo sviluppo essi tentano di risolvere.

### Molteplici ambienti

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

- Produzione: l'ambiente in cui l'applicazione viene utilizzata dagli utenti finali, in cui il codice è stabile e privo di bug (si spera).
- Test o UAT: in cui vengono testate le nuove funzionalità non ancora rilasciate, il codice è meno stabile e contiene molteplici bug
- Locale: installati sulle macchine di ogni sviluppatore, dove il codice viene modificato continuamente, l'ambiente è molto instabile e il codice spesso neanche compila.

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

Ad esempio: viene risolto un bug e la fix deve essere mandata in produzione il prima possibile, però i files modificati del codice sorgente contengono anche parti di funzionalità nuove che erano nell'ambiente di test ma non vanno mandate in prod perché non ancora pronte. È molto difficile tracciare quali parti del codice vanno effettivamente tenute e preparare quindi i file per la produzione, ci si mette molto tempo e si rischia di perdersi dei pezzi per strada.

### Molteplici sviluppatori

Solitamente i team sono composti da più sviluppatori che lavorano in parallelo sulla stessa versione iniziale del codice: mentre Alice sta lavorando su una nuova funzionalità, Bruno sta fixando un bug presente nell'ambiente di test e Carla sta migliorando la sicurezza del codice utilizzato per effettuare il login. Quando finiscono si accorgono di aver effettuato modifiche diverse agli stessi file, come fanno ad unire i loro sviluppi per mandarli nell'ambiente di test?

## Git

Git è un sistema di versionamento open source creato da Linus Torvalds nel 2005 per gestire il versionamento del kernel Linux ed è ad oggi il VCS più utilizzato in assoluto (87% degli sviluppatori al 2018).

Esso permette di:

- Tracciare ogni modifica ai file.
- Creare diverse versioni dello stesso progetto, potendo lavorare su ciascuna di esse parallelamente.
- Risolvere i conflitti tra modifiche simultanee ai files.
- Conservare su un server centrale l'intera cronologia di sviluppo e fare in modo che gli sviluppatori possano scaricarne una copia locale, permettendo di lavorare offline in completa autonomia.
- Scegliere una versione e caricare istantaneamente uno snapshot dei file dell'intero progetto in un qualunque punto della sua cronologia.
- Invertire una qualsiasi modifica, anche mantenendo quelle successive.

## 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 di siti che offrono servizi di repository git: 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, può eseguire qualsiasi modifica sui dati o sulla cronologia localmente; la "sincronizzazione" tra locale e remoto avviene solo manualmente a discrezione dello sviluppatore, che può scaricare dati (attraverso comandi come fetch e pull) o caricarli (push).

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

- Aprire una 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 contiene tutto l'occorrente per versionare il contenuto della cartella.

Una volta 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 è uno dei concetti principali di git, rappresenta il salvataggio (snapshot) dello stato dei file modificati, aggiunti o eliminati in un preciso istante della repository.

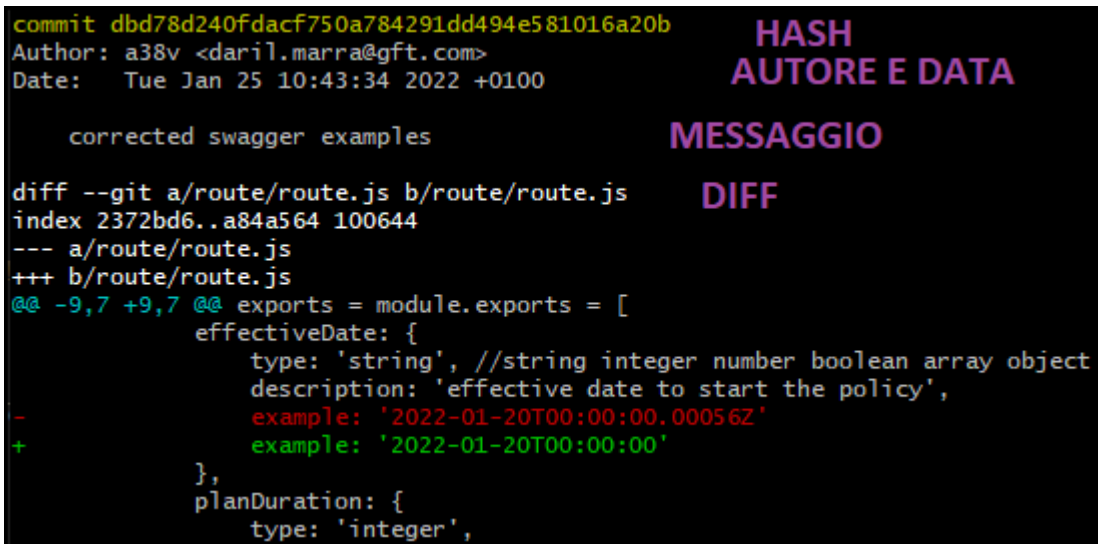
Essi risolvono il problema di tracciare ogni modifica ai file e sono i mattoni fondamentali della cronologia del progetto.

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

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

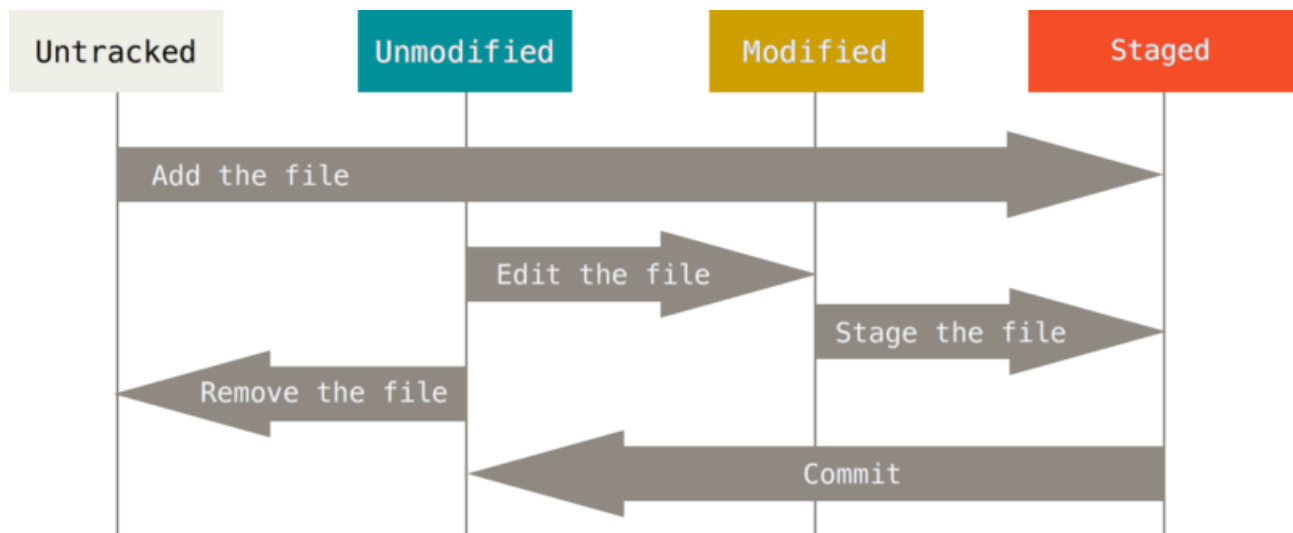
diff --git a/route/route.js b/route/route.js
index 2372bd6..a84a564 100644
--- a/route/route.js
+++ b/route/route.js
@@ -9,7 +9,7 @@ exports = module.exports = [
    effectiveDate: {
      type: 'string', //string integer number boolean array object
      description: 'effective date to start the policy',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

## Stato dei files

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

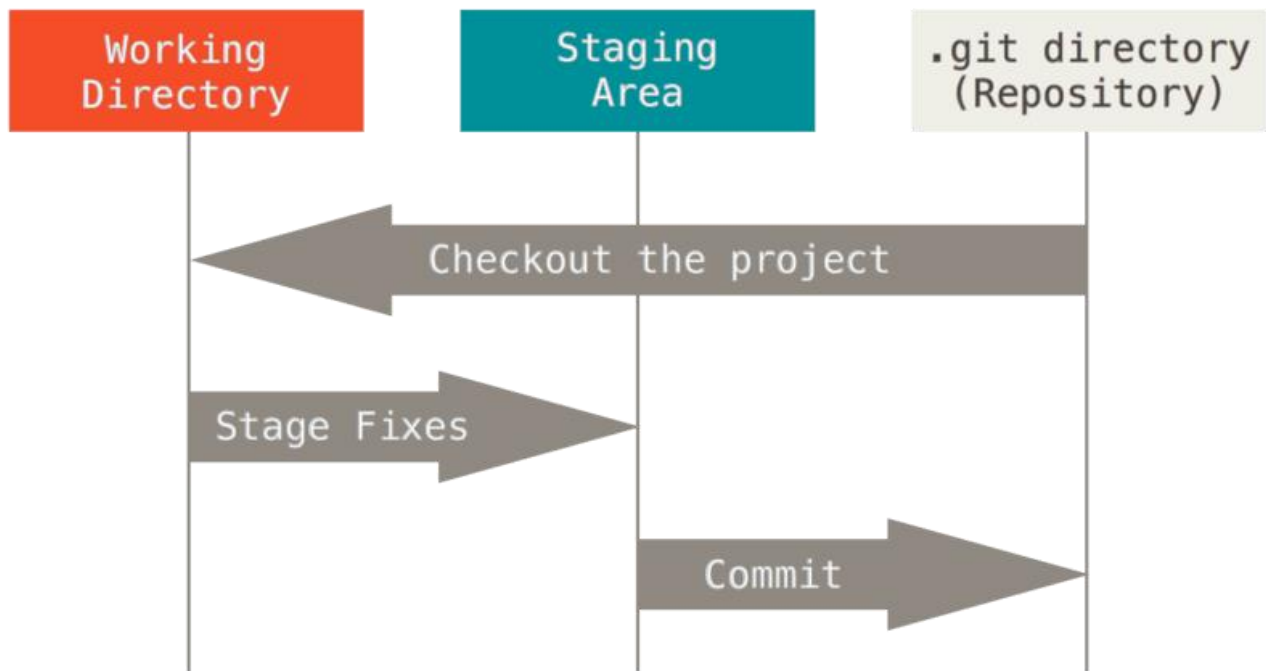
1. **MODIFIED:** files modificati la cui modifica non è ancora stata "committata" all'interno della repository
2. **STAGED:** i files modificati possono essere marcati per essere inseriti all'interno del prossimo snapshot che verrà creato col commit, quando succede i file passano in stato staged
3. **COMMITTED:** 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.



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



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 basta lanciare 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.

```
338v@ITPL009937 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)
    AGRI-1174.patch
    Commenti_patch
    HFM_patch
    PolicyCenter.bat
    Problema_date_patch
```

STAGED

MODIFIED

UNTRACKED



## 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 molteplici branch possono puntare allo stesso commit
- I branch permettono di creare più versioni diverse del progetto (controllo delle versioni) che possono essere destinati ad ambienti differenti
- I branch permettono agli sviluppatori di lavorare contemporaneamente sul codice
- Un branch può essere unito (mergiato) a un altro branch portando così su quest'ultimo tutte le modifiche fatte sul branch mergiato, sincronizzandoli

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.

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 (il branch sul quale siamo posizionati sarà evidenziato in verde):

> **git branch --all**

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.

Il checkout è praticamente istantaneo e ci permette facilmente di lavorare su più versioni, ad esempio lavorare su un branch che contiene sviluppi effettuati da un collega per aggiungere funzionalità correlate, oppure verificare il codice del branch di produzione per risolvere un bug, oppure di “viaggiare nel tempo” visualizzando lo stato del progetto ad un determinato commit.

Il branch corrente dell’area di lavoro è chiamato HEAD. HEAD non è altro che un puntatore all’ultimo branch di cui è stato effettuato il checkout, ma può anche puntare ad un commit invece che ad un branch (se si fa il checkout del commit): in quel caso il progetto è in uno stato chiamato “detached head” nel quale non si possono eseguire alcune operazioni quali commit finchè non si crea un branch corrispondente.

Per effettuare il checkout e posizionarci su un altro branch:

**> git checkout *nome\_branch***

Per fare il checkout di un determinato commit (il cui hash può essere recuperato con **> git log**):

**> git checkout *hash\_del\_commit***

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 *branch\_o\_commit -- cartella1/cartella2/file1.txt cartella3/cartella4/file2.txt***

## Operazioni remote

Finora abbiamo visto comandi che ci consentono di lavorare sulla nostra repository locale, ma non abbiamo ancora visto come interagire con la repository remota; prima di fare ciò dobbiamo poter distinguere tra branch locali e remoti.

- 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.

Vedremo ora alcuni comandi che lo sviluppatore può effettuare per sincronizzare i suoi branch locali con quelli remoti.

### Fetch

Fetch è un'operazione di download dalla repository remota.

- Tutti i nuovi branch presenti nella repository remota vengono scaricati nella 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

Push rappresenta l'upload sulla repository remota del branch.

È possibile *pushare* un branch nuovo, ed esso verrà aggiunto alla repository remota, oppure un branch già esistente in modo da aggiornare la sua versione remota con i nuovi commit effettuati su di esso in locale.

La prima volta che si esegue il push dello specifico branch si usa il comando:

> **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**

### Merge

Merge aggiunge ad un branch il contenuto di un altro branch.

Tipicamente un progetto ha un branch principale sul quale vengono riportati tutti gli sviluppi: gli sviluppatori lavorano separatamente sui propri branch i quali 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 locale corrente sia *dev*, questo comando andrà a mergiare *nome\_branch\_da\_mergiare* nel branch *dev*, incorporandone tutti i commit.

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

## Pull

Pull è un 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 remoto 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**

Anche in questo caso come con `push` possiamo aggiungere il flag `-u` per collegare il branch remoto a quello locale in modo da utilizzare semplicemente i comandi **git pull** e **git push** le volte successive.

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

Quando si effettua un merge (o un pull), git tenta di conciliare automaticamente le modifiche eseguite ai file, ma a volte capita che un file venga modificato nello stesso punto in commit diversi dei branch che si tenta di mergiare; in questo caso il merge “fallisce” e viene chiesto allo sviluppatore di concludere il merge manualmente risolvendo i conflitti.

Sarà quindi possibile vedere le modifiche presenti sui due branch, scegliere quali mantenere o scartare ed eventualmente modificare ulteriormente il file per permettere alle modifiche di coesistere, dopodichè vanno aggiunti i file alla staging area (con **> git add .**) e completato il commit di merge (**> git commit**).

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 sul sistema utilizzato 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. Non c'è differenza tra Merge Request e Pull request, semplicemente alcuni sistemi la chiamano in un modo e altri in un altro.

Il branch, prima di essere mergiato, viene spesso sottoposto a una code review, dove un collega più esperto verifica il codice: egli 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 sarà aggiornata con le ultime correzioni e il reviewer potrà mergiare il branch dello sviluppatore nel branch principale.

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

## .gitignore

All'interno di un progetto succede che alcuni files non debbano essere condivisi, ad esempio:

- Files generati dal processo di build
- Files di configurazione dell'ambiente, che contengono proprietà diverse in base ai vari ambienti di test, prod e per ogni ambiente locale degli sviluppatori
- Files che contengono informazioni segrete, quali credenziali dei servizi integrati nel progetto (API keys, credenziali del database, ecc..)

Per questo git mette a disposizione una funzionalità per ignorare tali files, attraverso il file `.gitignore`. `.gitignore` è un file che viene inserito nella cartella base di ogni progetto e 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: *\*.x/sx*)
- 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.

## Altri comandi utili

- **git cherry-pick *hash\_del\_commit*** eseguito passando come argomento l'hash di un commit presente su un altro branch, riporta questo commit (con le modifiche presenti in esso) all'interno del branch attivo; è quindi simile al merge ma riporta un solo specifico commit invece dell'intero branch e come merge può portare a conflitti, risolvibili con lo stesso metodo
- **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 reset --hard *origin/nome\_branch*** ripristina il working tree esattamente allo stesso stato del branch remoto selezionato, cancellandone ogni eventuale commit aggiuntivo ed eliminando ogni modifica presente nel working tree. Eseguire sempre prima un **fetch** in modo da allinearsi al remoto
- **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
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