

# Strumenti di versionamento: GIT

Rev 1.4 del 24/01/2025

Introduzione agli Strumenti di versionamento .....	2
Introduzione a GIT .....	4
Gli snapshot .....	5
Lo stato dei files (GIT workflow) .....	6
<b>Principali comandi git</b> .....	6
git config .....	7
git init .....	7
readme.md .....	7
.gitignore .....	8
git clone .....	9
git add .....	9
git commit .....	9
git restore .....	10
git remote, git push e git pull .....	10
<b>Comandi per la manipolazione dei singoli files</b> .....	12
git status .....	12
Cancellazione di un file .....	13
Spostamento di un file .....	13
git diff .....	14
Cronologia delle modifiche .....	14
<b>Creazione e Gestione dei branch</b> .....	15
git branch .....	15
git merge .....	16
Tag e numeri di versione .....	18
Alias dei comandi .....	18
git stash .....	19
git rebase .....	19
<b>Github</b> .....	20
Modalità di autenticazione da remoto .....	20
Creazione di un token e impostazione delle credenziali di accesso .....	21
Creazione e Gestione di un repository su GitHub .....	23
Navigazione del Repository .....	24
Impostazioni del Repository .....	25
Pull request .....	26
Utilizzo di git all'interno di Visual Studio Code .....	27
Prontuario Veloce .....	287

## Introduzione ai Sistemi di versionamento (VCS)

Un sistema di versionamento (**VCS** = Version Control System) è un sistema che serve a tenere traccia delle modifiche apportate ad un progetto in modo da consentire ad un team di collaborare insieme e risparmiare tempo. Un sistema di versionamento traccia la storia delle modifiche apportate ad un progetto, chi le ha fatte, quando e perché.

Utile ad esempio per poter risalire ad un intervento causa di un malfunzionamento, annullare le modifiche e ripristinare la precedente release di un file o di un progetto nel suo insieme.

In linea generale, si tende ad associare il controllo di versione allo sviluppo di applicazioni, ma è possibile applicare il versioning anche a progetti di natura differente, come per esempio la documentazione relativa ad un prodotto o ad un linguaggio, le specifiche associate a uno standard, una cronologia, un layout grafico (si pensi per esempio all'evoluzione di un logo) e, nel complesso, qualsiasi file sia suscettibile di modifiche.

I sistemi di versionamento possono essere :

- locali (**VCS** semplici)
- centralizzati (**C-VCS**)
- distribuiti (**D-VCS**)

### 1. Controllo di versione locale

---

E' la forma più essenziale e primitiva di versioning; si immagini uno sviluppatore che crea un'applicazione, effettua la presentazione o la messa in produzione di quest'ultima e poi, magari su suggerimento del committente, effettua delle modifiche introducendo aggiornamenti e correzioni. In questo caso le diverse versioni del progetto potrebbero essere salvate in differenti cartelle rinominate, ad esempio associando la data dell'ultima modifica effettuata al nome del progetto.

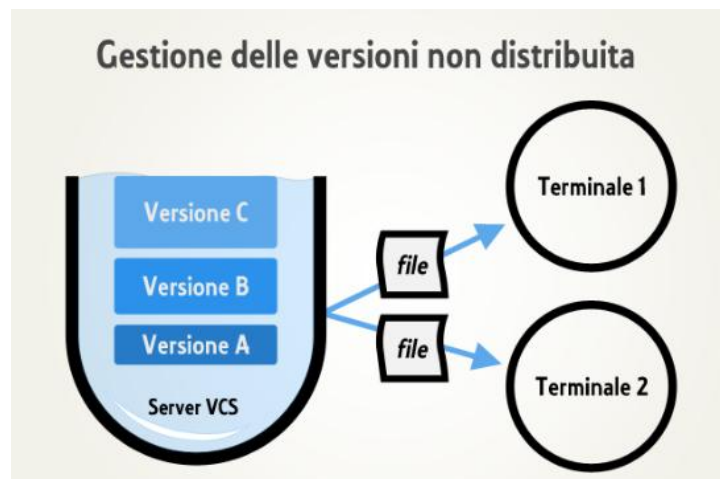
Questo approccio non necessita di un software per la gestione delle diverse release, ma diventa complicato nel caso di progetti articolati coinvolti da numerose modifiche. Tenere traccia degli interventi effettuati diventerebbe complesso con il rischio di perdere molte informazioni con il passare del tempo e l'accumularsi degli interventi eseguiti. Per questo motivo sono nate le prime soluzioni per il versioning come per esempio **RCS** (*Revision Control System*), che utilizza un database destinato a registrare le modifiche apportate ai file e delle patch che memorizzano le differenze tra le revisioni in modo da semplificare le operazioni di ripristino. Questi sistemi facilitano il controllo di versione locale ma non offrono strumenti per lo sviluppo in ambito collaborativo.

### 2. Controllo di versione centralizzato

---

Ad esempio **Subversion**. I sistemi centralizzati operano sulla base di un singolo server a cui viene affidato il compito di ospitare tutte le varianti dei progetti sottoposti a versioning. Tale server permetterà il download dei file da parte degli utenti interessati a partecipare allo sviluppo, la creazione delle fork (versioni derivate dal sorgente di un progetto originale) o semplicemente l'utilizzo dell'applicazione.

I sistemi centralizzati offrono degli indubbi vantaggi rispetto a quelli per il controllo di versione locale, soprattutto in un contesto che prevede la partecipazione di più soggetti sarà più facile tenere traccia delle modifiche apportate da altri. Inoltre gli amministratori hanno la possibilità di accordare ad utenti differenti diversi privilegi di accesso e scrittura ai file.

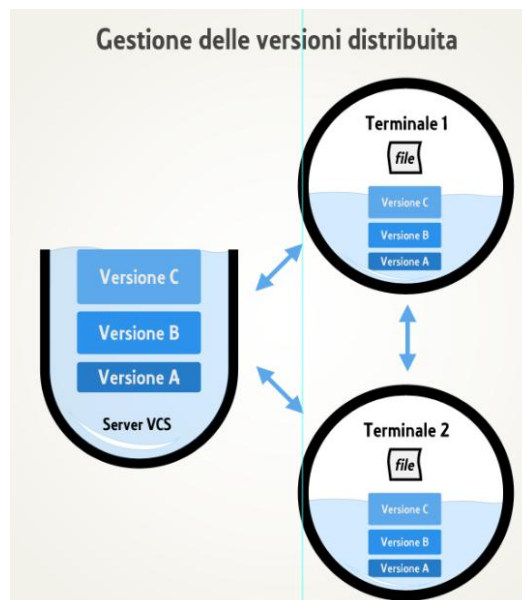


I CVCS presentano però un limite : in caso di malfunzionamento dell'unità centrale tutti i file risulteranno irraggiungibili e nessuno dei collaboratori coinvolti potrà operare su di essi. Ancor peggiore l'eventualità di un danneggiamento con conseguente perdita dei dati. Di qui l'esigenza di sviluppare delle alternative basate sulla decentralizzazione (o distribuzione) del controllo di versione.

### 3. Controllo di versione distribuito

Con il termine **repository** si intende l'insieme di files e cartelle associati ad un progetto, **compreso tutto lo storico delle modifiche**.

I repository rappresentano il cuore dei DVCS, i quali prevedono la possibilità che il client possa eseguire copie complete dell'intero repository. Questo grazie ad un'apposita funzionalità che consente di clonare un intero repository da una cartella locale o da un server remoto e viceversa.



La possibilità di lavorare localmente offline presenta un vantaggio non indifferente: in mancanza di una connessione si potrà operare localmente su un progetto affrontando tutte le fasi del versioning fatta eccezione per quella dell'upload, che potrà essere completata non appena sarà disponibile un nuovo collegamento.

Inoltre, in caso di problemi da parte dei server, il repository presente in un client qualunque potrà essere utilizzato per il ripristino.

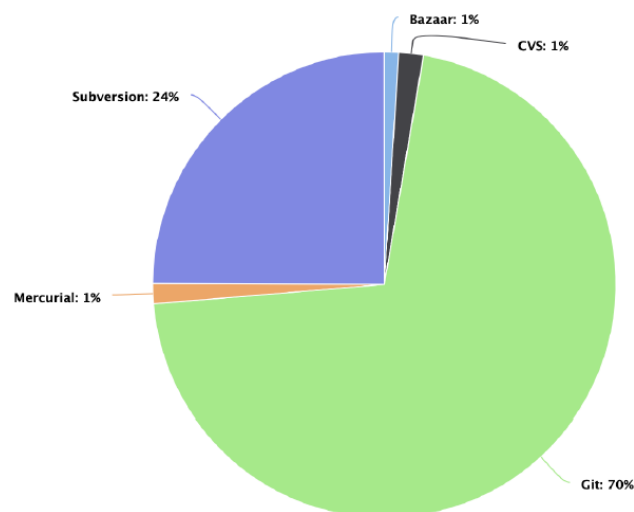
## Introduzione a GIT

GIT è un **D-VCS** proposto da Linus Torvalds nel 2005.

Il kernel Linux per lungo tempo è stato gestito in ambito collaborativo tramite lo scambio di semplici package ai quali venivano aggiunte delle patch per le correzioni e gli aggiornamenti per il passaggio alle nuove release. Tale sistema divenne eccessivamente macchinoso man mano che l'architettura del Kernel diveniva più articolata.

Da qui l'esigenza di adottare un DVCS e la nascita di GIT

La parola "**git**" significa **idiot**, cioè molto semplice, a prova di stagista. Git è distribuito gratuitamente e copre oggi (novembre 2018) il 70% del mercato del versioning.



Negli anni passati è stato molto utilizzato Mercurial, oggi quasi completamente abbandonato per ragioni di prezzo e anche perchè non è più stato aggiornato.

Sui sistemi MAC e UNIX git viene installato insieme al SO. Su windows si può installare "**git for windows**" il quale, tramite una cartella nascosta .git, gestisce tutti i files di versionamento, consentendo in qualunque momento il ripristino di una versione precedente.

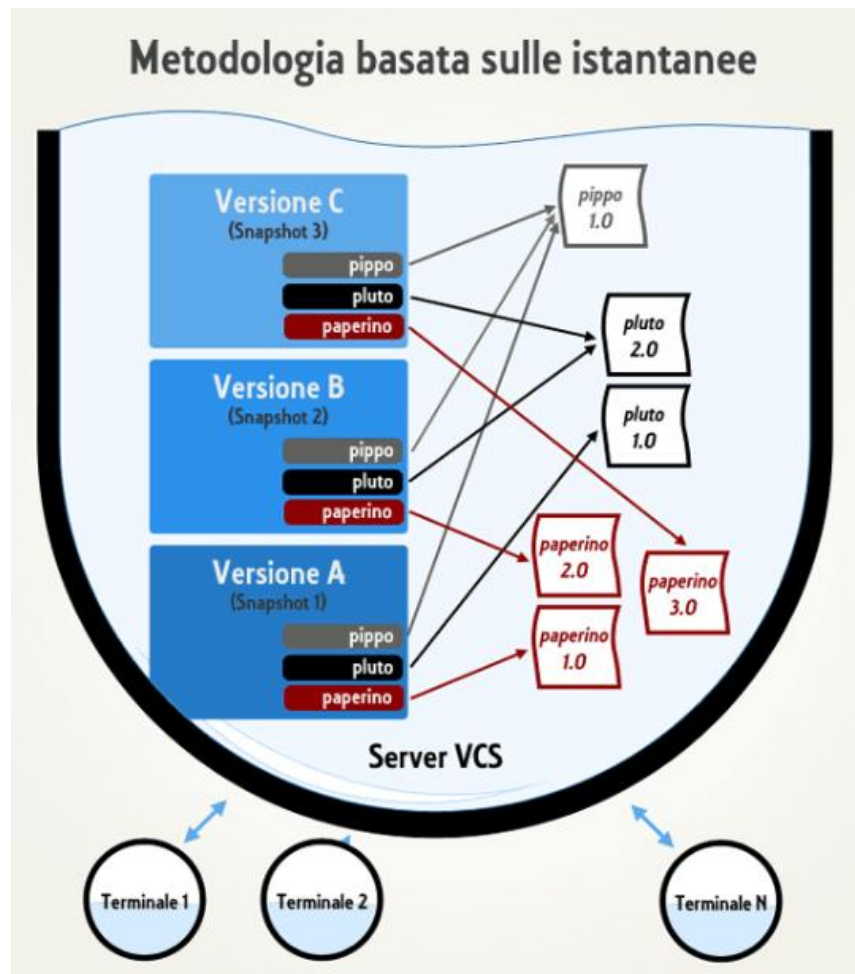
## Gli snapshots

Git salva i files come delle **istantanee**, o snapshots. Cioè Git non gestisce le differenze tra le varie versioni, ma "fotografa" i vari stati di avanzamento del progetto.

Quando si memorizza lo stato di un progetto il sistema crea una snapshot di tutti i file al momento corrente.

La varie snapshots sono **reversibili**, nel senso che in ogni momento è possibile risalire dalla snapshot al file originale e sono salvate all'interno della sottocartella Objects

Nel caso in cui **un singolo file** non dovesse subire delle modifiche nelle versioni successive (nell'esempio il file denominato **pippo.txt**), il DVCS non provvederà a salvarlo in ogni versione successiva, ma si limiterà a definire un collegamento con il salvataggio precedente.



Da questo punto di vista git esegue un backup **incrementale**. Ogni volta **salva solo i file variati rispetto alla revisione precedente**. Se un file non è cambiato lo andrà a prendere nella revisione precedente e così via.

### Lo stato dei files (GIT workflow)

I file di un progetto durante lo sviluppo locale possono assumere tra stati differenti

- **Committati** : i file fisicamente salvati come snapshot nel database locale
- **Modificati** : i file correnti modificati ma non ancora approvati per il commit
- **In stage** : si tratta di file modificati e già approvati per il commit ma non ancora committati, e che quindi verranno inclusi nella prossima commit

Ad ognuno di questi stati corrisponde una specifica sezione di un progetto git:

- **Directory git**: contiene tutti i file committati relativi al progetto, cioè inizialmente tutte le informazioni copiate in fase di clonazione, poi via via le varie snapshot committate relativamente alle versioni successive.
- **Working Directory** contiene la **copia** di lavoro di una determinata versione del progetto.
- **Area di stage** : Definita anche come **HEAD** o **Indice** rappresenta una sorta di entità intermedia tra la **directory di lavoro** e la **directory .git**, preposta a contenere gli snapshot dei files "approvati" per il commit ma non ancora committati

In sostanza i file modificati subiscono un doppio passaggio: prima devono essere "approvati" con l'aggiunta degli **snapshots** nell'Area di Stage e solo dopo committati. Questo doppio step evita di dover eseguire un commit ogni volta che si modifica un file



- Un file modificato e aggiunto nell'area di stage si dice "**in stage**",
- una release presente nella directory di Git si dice "**committata**",
- una versione estratta, modificata ma non "in stage" si dice "**in lavorazione**".

Per ogni file, oltre allo snapshot, Git utilizza un sistema di controllo dei file basato sul **checksum**. Ad ogni snapshot di ogni file viene associato un hash non reversibile (cioè una stringa di lunghezza predefinita) prodotto tramite algoritmo di cifratura **SHA-1** che restituisce stringhe di 40 caratteri. Da questo checksum git capisce immediatamente se il file è stato modificato e agisce di conseguenza. Molto veloce..

Tali stringhe inoltre consentono di eseguire delle **verifiche di integrità**; un elemento viene considerato integro se corrisponde al checksum ad esso associato.

## Principali comandi git

In fase di installazione accettare tutte le impostazioni proposte come default.  
Dal **2024** (versione 2.3.0) insieme a git viene installato anche il nuovo Git Credential Manager per facilitare la gestione dell'accesso da remoto.

### git version

Restituisce il numero di versione di git. 2.47 a gennaio 2025

### git update

esegue l'aggiornamento di git all'ultima versione disponibile.  
Comando sostituito nel 2024 da **git update-git-for-windows**

### creazione di un nuovo progetto : git init

Creare un nuovo progetto Git significa trasformare la nostra generica cartella di lavoro in un repository per il versioning. Il comando da utilizzare è **git init**. Lanciato da terminale nella cartella di lavoro corrente (contenente i files di progetto), la 'trasforma' in un **repository GIT** creando una sottocartella nascosta **.git** preposta a contenere l'intero repository e la sua struttura. Se si volesse creare una copia di backup del nostro repository sarebbe sufficiente duplicare la cartella **.git**.

Ancor prima di lanciare il comando **git init** è bene creare manualmente, sempre nella cartella di lavoro, due files testuali molto importanti:

- **readme.md**
- **.gitignore**

### git clone

Invece di partire da un progetto locale in fase di sviluppo, si può clonare un repository già presente in locale oppure in rete e clonarlo tramite il comando **git clone**

Clone da un server di rete (l'estensione **.git** può anche essere omessa)

**git clone https://github.com/robertomana/myScada.git**

Clone da una cartella locale

**git clone "c:\myFolder\myProject"**

In corrispondenza del clone la **working directory** viene automaticamente sempre impostata sul **branch main**

### Il file readme.md

Il file **readme.md** è un file testuale di introduzione e spiegazione del progetto. Deve essere scritto utilizzando un cosiddetto linguaggio di markdown. Il **Markdown** (wikipedia) "*è un linguaggio di markup per la formattazione di pagine testuali con una sintassi del testo semplice progettata in modo che possa essere convertita in HTML e in molti altri formati usando un tool omonimo.*"

Si tratta di un linguaggio più leggero e con minor potenzialità rispetto all'html, ma sicuramente molto più veloce nella scrittura dei documenti.

Ad esempio gli headings, simili ai tag **<h>** dell'html, sono semplicemente preceduti dal simbolo **#**

```
#      heading level 1 (<h1>)  
##     heading level 2 (<h2>)  
###    heading level 3 (<h3>)
```

Esempio di file **readme.md**:

```
###  Hi there, I'm a student of IIS Vallauri  
##   nodejs mail server project
```

## Il file .gitignore

E' un file testuale contenente, uno per riga, un elenco di files/cartelle che non devono essere sottoposti a versioning, perché non significativi ai fini del progetto o facilmente replicabili. Ad esempio si possono ignorare i files.log, i files.tmp e l'intera cartella node\_modules (con tutti i suoi files e sottocartelle).

```
*.log
*.tmp
node_modules    // intera cartella node_modules
```

## Struttura del file .gitignore

Non tutti i file appartenenti ad un progetto sono utili per le attività di sviluppo e versionamento, si pensi per esempio ai log, alle librerie, agli eseguibili. Tutti i DVCS consentono di definire una **lista di esclusione** rappresentata dal file **.gitignore**

- Quanto indicato nel file .gitignore ha effetto su tutte le directory di un progetto, a meno che esse non presentino a loro volta un proprio file .gitignore
- I files indicati all'interno di .gitignore saranno ignorati da qualsiasi comando git. Non ha però nessun effetto sui file già tracciati o che sono al momento in stage in attesa di commit. Se si modifica il file .gitignore, è sempre bene parallelamente ripulire lo stage con il comando **git rm -r --cached .** (ripulisce l'intera head in modo ricorsivo) e poi fare **git add** e **git commit**
- Al fine di evitare inutili complicazioni, le voci contenute all'interno del file .gitignore **NON devono contenere degli spazi !**

## Sintassi di scrittura

Il file gitignore è costituito da una **sequenza di righe** ognuna delle quali indica un preciso file da "ignorare". Ogni riga in realtà è costituita da un **regular patterns** che consente ad esempio l'utilizzo del carattere jolly \*.

# indica una riga di commento

### Esempio

```
# ignora tutti i file con estensione .log e .tmp
*.log
*.tmp
# ignora la cartella node_modules e tutto il suo contenuto
node_modules
```

### Espressioni regolari

All'interno di .gitignore è anche possibile utilizzare le regular expression:

```
# ignora tutti i file che terminano con x o y
*[xy]
# ignora tutti i file che terminano con il simbolo ~
*~
# ignora tutti i file HTML
*.html
# tranne index.html
!index.html
```



## git config

Il comando **git config --list**

visualizza un elenco di tutte le impostazioni relative all'installazione locale sul proprio PC

```
git config user.name      // nome utente repository corrente
git config user.email     // email repository corrente

// per modificare l'utente del repository corrente
git config user.name "Nuovo Nome"
git config user.email "nuova.email@example.com"

// per modificare l'utente a livello di macchina
git config --global user.name "Nuovo Nome"
git config --global user.email "nuova.email@example.com"
```

## git add

### git add filename

salva nella HEAD una snapshot dello stato attuale del file indicato, snapshot pronta per essere salvata nel repository in corrispondenza del prossimo commit.

In realtà di per se il **git add** non sarebbe indispensabile perché, quando si esegue un commit, c'è l'opzione **-a** che consente di controllare automaticamente tutti i files che hanno subito delle modifiche e salvarli nella head e quindi nel repository.

Il problema è che ogni volta che si crea / aggiunge un nuovo file al progetto, **il nuovo file non viene automaticamente tracciato**. Se lo sviluppatore desidera includerlo nel tracciamento, deve eseguire esplicitamente un comando **git add**. Lo scopo è quello di impedire l'inclusione involontaria di componenti non strettamente necessari.

Però poi spesso si finisce per dimenticare questo comando per cui, per questo e per altre ragioni che si vedranno in seguito, è bene eseguire SEMPRE il comando **ADD subito prima** del comando di commit. Esempi:

```
git add server.js // si possono indicare più files separati da spazio
git add static/*  // tutti i files contenuti nella cartella static

git add .         // tutti i files in lavorazione che hanno subito modifiche
git add -A        // uguale al precedente
```

## git commit

### git commit -m "Initial Project Version 0.1"

Il commit provvede a caricare tutti i contenuti dello stage nel repository locale dei files approvati. Vengono salvate tutte le **variazioni** rispetto al commit precedente.

L'opzione **-m** è obbligatoria (senza l'opzione il commit non viene eseguito) e consente di definire un *log message* di descrizione del commit. Questo messaggio verrà salvato all'interno del file COMMIT\_EDITMSG. Al suo interno dichiarare le modifiche solo ad alto livello, senza superare i 50 caratteri e scrivendo in inglese. All'inizio del *log message* si inserisce di solito una delle seguenti parole chiave che spiegano immediatamente il tipo di intervento effettuato :

**feat** = nuova funzionalità  
**fix** = correzione errore  
**chore** = piccolo aggiustamento  
**doc** = aggiunto documentazione

L'opzione **-a** (all) provvede a salvare nel database, oltre al contenuto dello stage, anche tutti i file modificati nella working directory, bypassando sostanzialmente il passaggio in stage. E come se prima eseguisse git add per tutti i file modificati. *Attenzione però che l'opzione -a 'committa' i files modificati, ma NON eventuali nuovi files, che devono essere aggiunti esplicitamente tramite il comando **git add**.*

L'opzione **--amend** consente di risolvere una problematica abbastanza frequente, quella di aver effettuato un commit prematuramente, dimenticando di includere alcune modifiche. L'opzione **--amend** fa sì che il commit corrente "sostituisca" completamente il commit precedente, senza creare una nuova versione. In corrispondenza del **--amend** viene anche aperto l'editor di testi associato a git che ripropone il messaggio di descrizione dell'ultimo commit consentendo una sua eventuale modifica.

Se il commit viene eseguito su un branch diverso da master, dopo il codice precedente occorre inserire il nome del branch a cui il commit si riferisce.

## git restore

**git restore<file>** consente di annullare le modifiche apportate al file nella working directory e ritornare all'ultima versione committata

**git restore --staged <file>** consente di annullare le modifiche che ritornano all'ultima versione committata. Viene invece mantenuta la versione attualmente in lavorazione nella working directory.

**git restore --staged <file>** ripristina sia lo stage sia la working directory.

## Gestione dei repository remoti: remote, push e pull

### Configurazione del server remoto

**git remote add** memorizza all'interno del repository locale l'indirizzo git del server remoto. Ogni progetto locale memorizza al suo interno l'origine del repository remoto su cui salvare. **git remote add** si aspetta due parametri:

- un nome di identificazione **locale** del server remoto (spesso indicato con il termine "**origin**")
- la URL del server GIT di destinazione.

Quando il progetto deve essere uploadato su più repository parallelamente, è possibile eseguire più comandi **remote add** utilizzando un identificativo diverso per ciascun server e poi eseguire fisicamente **due push** indipendenti.

```
git remote add origin https://github.com/robertomana/myScada.git
git remote add heroku https://git.heroku.com/myScada.git
```

**git remote** senza parametri visualizza la lista completa dei repository associati. In caso di repository clonati, il comando **git remote** restituisce in output almeno un server remoto (quello di provenienza), a cui assegna automaticamente il nome origin.

**git remote -v** visualizza la lista completa dei repository associati, **ciascuno con la propria URL**. Ogni URL viene riportata due volte, la prima per il download (**fetch**), la seconda per l'upload (**push**) in modo da poter eventualmente utilizzare 2 server diversi.

**git remote show origin**  
Visualizza la URL del singolo repository indicato

**git remote remove origin**  
Se si ha necessità di reimpostare una url, occorre prima rimuovere il link corrente e dopo reimpostarla

**git remote rename origin heroku**  
consente di modificare il nome di identificazione locale di un server remoto

---

## Il comando GIT PUSH

Dopo aver configurato origin, si può eseguire la push del branch corrente sul server remoto. In locale si lavora sempre SOLO su un singolo branch, mentre sul server possono esserci più branch differenti su cui stanno lavorando persone differenti. Il nome del branch principale dipende dalla struttura del repository. Github in passato assegnava al branch principale il nome **master**. Ora assegna invece il nome **main**

**git push origin master** Carica il branch locale sul server indicato da **origin** all'interno del branch indicato dal 2° parametro (**master**).

**git push -u origin master** L'opzione **-u** memorizza origin master come parametri di git push, per cui la prossima volta che si eseguirà git push senza alcun parametro, automaticamente verranno invocati il server ed il branch memorizzati con l'ultima esecuzione di git push -u.

**git push** senza parametri esegue l'ultimo push eseguito con l'opzione -u. Per cui è sufficiente eseguire il comando completo soltanto in corrispondenza del primo push

**git push --force** sovrascrive completamente il repository remoto

**Notare che** il comando **git push** opera sempre soltanto su un singolo repository remoto e su un singolo branch. Se si vuole eseguire il push su più repository oppure su più branches occorre eseguire più push successive tutte in forma completa.

---

## Il comando GIT PULL

Uguale opposto al precedente. Dopo aver settato origin, eseguire **git pull origin master**. Esegue il download del branch indicato (master) dal server indicato (origin) utilizzando il commit più recente presente sul server.

L'opzione **-u** non è disponibile ed è sostituita dalla seguente:

**git branch --set-upstream-to=origin/main master**  
dove **origin/main** rappresentano l'origine di destinazione ed il branch di destinazione, mentre l'ultimo parametro master indica il nome dell'attuale branch locale che ad oggi, gennaio 24, continua a chiamarsi master e non main.

In alternativa, la prima volta che si scarica un progetto dalla rete si può eseguire il comando **clone** molto più rapido con posizionamento automatico sul branch main.

**Nota:** Anche il comando **git pull** opera su un singolo branch.

Nel caso della pull però, git verifica anche se sul server esistono nuovi branch non presenti in locale, nel qual caso li crea anche in locale senza però aggiornarne il contenuto, che dovrà essere aggiornato tramite pull successive.

### Sequenza dei comandi al termine di una sezione significativa

---

Dopo aver configurato l'indirizzo del server remoto mediante il comando **git remote** ed aver eseguito un primo push, ogni volta che si termina una sezione significativa di lavoro, la si può salvare sul repository remoto mediante la sequenza di comandi:

```
git add .  
git commit -a -m "changes description"  
git push
```

Per scaricare l'ultima versione dal server :

```
git pull origin main
```

### Gestione dei Conflitti

---

Se si fanno delle modifiche in locale e nel frattempo qualcun altro ha aggiornato il server con una push, nel momento in cui si esegue un commit locale e si tenta di fare una push si genera un errore che avvisa che il repository sul server è cambiato e bisogna fare prima una pull.

Facendo la pull, **se le modifiche locali non interferiscono con le modifiche fatte sul server** (ad esempio perché applicate a due procedure diverse) git provvede ad eseguire automaticamente il merge per cui, sul repository locale mi troverò sia le modifiche apportate in locale sia quelle provenienti dal server. Se invece le modifiche vanno in conflitto su una stessa porzione di codice, allora occorre risolvere manualmente tutti i conflitti analizzando tramite `git diff` le differenze fra i due files.

## Comandi per la manipolazione dei singoli files

### Il comando git status

---

**git status** consente di verificare lo stato dei file relativi al progetto.

- Se lo si lancia senza aver apportato modifiche dopo l'ultimo commit, questo comando produrrà una notifica del tipo *nothing to commit, working tree clean*, che testimonia come la *working directory* sia "pulita" al momento corrente, proprio perché, in seguito ad un commit, nessuno dei file sottoposti a tracciamento ha subito ancora alcuna modifica.
- Se invece ci fossero dei files modificati dopo l'ultimo commit, oppure in staging, l'output notificherebbe un messaggio con le scritte :  
"Changes to be committed" (verde) per i file in staging  
"Changes not staged for commit" (rosso) per i file modificati in lavorazione.

### **File non tracciati**

Come detto un nuovo file viene impostato come non tracciato(**untracked**).

Nel caso di file non tracciati, `git status` provvederebbe ad elencarli. In caso contrario si ha la sicurezza che tutte le componenti del progetto risultano tracciate.

La creazione di una cartella vuota non crea problemi. Il versioning è fatto solo sui files

### **Clonazione**

In seguito alla clonazione tutti i file prelevati dal repository scelto vengono classificati da subito come tracciati e non modificati.

### **Modifica di files staged**

Può anche capitare di apportare ulteriori modifiche ad un file che, a seguito di una precedente modifica, è già stato aggiunto allo stage, prima che quest'ultimo sia stato coinvolto in un commit. In corrispondenza del comando `git status` questo file comparirebbe nello stesso tempo :

- sia in verde, con la notazione "Changes to be committed"
- sia in rosso, con la notazione "Changes not staged for commit"

In questo caso occorre ripetere il comando `git add` perché, altrimenti, a passare nel repository sarà la versione del file attualmente in stage, e l'ulteriore modifica rimarrà nella working directory ma NON verrà committata.

Viceversa richiamando `git add` la nuova versione del file andrà a sovrascrivere la versione precedente nella head, che dunque non verrà mai committata.

Questo è un altro motivo per cui conviene sempre fare `git add` prima del commit.

### **Cancellazione di un file**

#### **git rm filename**

elimina il file dalla cartella corrente e salva la versione corrente come **deleted** in stage, in modo che il file venga definitivamente eliminato in corrispondenza del prossimo commit.

- In questo modo si tiene traccia dell'ultimo contenuto del file prima della sua definitiva cancellazione.
- Se il file non è variato rispetto all'ultimo commit oppure non si ritiene necessario dover memorizzare le ultime variazioni, questo comando può anche essere tralasciato

`git rm --cached filename` elimina il file dalla head

`git rm -r --cached .` ripulisce l'intera head (-r sta per -recursive, cioè cartelle e sottocartelle).

**Nota:** Il comando `git rm` supporta l'utilizzo di patterns e wildcards per l'eliminazione delle risorse. I due esempi seguenti mostrano rispettivamente:

- come cancellare tutti i file .txt
- come cancellare tutti i file presenti nella cartella docs ed aventi estensione .txt

```
git rm \*.txt
git rm docs/\*.txt
```

In entrambi i casi si osservi l'utilizzo del backslash prima dell'asterisco, necessario per il corretto funzionamento dell'istruzione.

## Spostamento di un file

Il rename / spostamento di un file sono considerati cancellazione + aggiunta. Se si sposta un file da una cartella all'altra o si rinomina un file, in entrambi i casi facendo `git status` si vedranno due file modificati: un file **deleted** ed un nuovo file **untracked**. Questo è un altro motivo per cui, prima del **commit**, fare SEMPRE `git add`

## il comando git diff

**git diff** <file>

- Se esiste una versione in stage, esegue un confronto fra il **file in lavorazione** e quello **in stage**
- Se NON esiste una versione in stage, esegue un confronto fra il **file in lavorazione** e l'ultimo **committato**

In entrambi i casi produce il seguente output:

- in rosso con davanti un - le righe del vecchio file eliminate o modificate
- in verde con davanti un + le righe del nuovo file contenenti le modifiche apportate o aggiunte ex novo
- in bianco eventuali sezioni intermedie non modificate

```
var securityMode = opcua.MessageSecurityMode.get("NONE");
-if (!securityMode)
-  throw new Error("Invalid Security mode , should be " + opcua.MessageSecurityMode.enums.join(" "));
-var securityPolicy = opcua.SecurityPolicy.get("None");
-if (!securityPolicy)
-  throw new Error("Invalid securityPolicy , should be " + opcua.SecurityPolicy.enums.join(" "));
+^M
var options = {
  securityMode: securityMode,
  securityPolicy: securityPolicy,
-  defaultSecureTokenLifetime: 40000
+  defaultSecureTokenLifetime: 40000, ^M
+  "nuovaAggiunta:"nuovoValore" ^M
}
```

**git diff --staged** visualizza le modifiche intercorse tra il file **in stage** e l'ultimo **commit**

## Cronologia delle modifiche

**git log** consente di visualizzare una cronologia completa dei commit effettuati e ordinati a partire dal più recente, ciascuno con il proprio messaggio di descrizione.

- l'argomento **-p** visualizza le differenze tra i diversi commit eseguiti. L'opzione **-p** può essere seguita da un valore numerico intero, per limitare il numero di commit visualizzati, ad esempio `git log -p -3` mostrerà in output i soli ultimi 3 commit.
- Siccome l'output è piuttosto lungo, lo si può redirigere su un file `commit.log` utilizzando l'operatore **>**.

## Creazione e Gestione dei branch

Quando si crea un nuovo repository, automaticamente viene creato un ramo predefinito che per default si chiama **master** (più recentemente **main**).

Se il repository si trova su un server di produzione (ad esempio heroku), non si può ovviamente lavorare su master perché le varie modifiche potrebbero causare delle interruzioni sul server o peggio dei malfunzionamenti.

I **branch** sono lo strumento che consente di definire delle diramazioni di sviluppo rispetto al ramo principale. E' anche possibile aprire più branch paralleli in modo da consentire a persone differenti lo sviluppo parallelo di funzionalità tra loro isolate (cioè indipendenti l'una dall'altra), a partire dalla medesima radice.

Un nuovo branch viene creato sulla macchina locale ed avrà un suo repository dedicato

### Il comando git branch

#### **git branch develop**

crea un nuovo branch locale avente il nome indicato ed avente come genitore l'ultimo commit eseguito. **In pratica l'ultimo commit del genitore viene copiato all'interno del database del branch.** Il branch attivo rimane però il master. Se poi sul master si dovessero eseguire altri commit, il branch **develop** continuerà a puntare al commit da cui è stato generato.

**git branch** eseguito senza parametri visualizza tutti branch del progetto corrente, ricolorando di verde (con un asterisco davanti) il branch attualmente attivo (che rimarrà attivo fino all'esecuzione di un nuovo git checkout).

#### **git checkout branchName**

Il comando checkout consente di navigare attraverso i vari branches. Consente cioè di selezionare il branch indicato che diventa la "working directory". Il comando **git checkout branchName** aggiorna i files della working directory con quelli presenti nell'ultimo commit del branch. I futuri commit saranno eseguiti sul branch e non più sul master. Ad ogni commit il ramo **develop** avanzerà di una posizione mentre master continuerà a puntare al commit nel quale si trovava prima del checkout.

Lanciando il comando **git status** prima del commit si vede chiaramente che i "changes to be committed" sono riferiti al branch **develop**.

**git checkout** eseguito senza parametri esegue un checkout del branch corrente. In pratica sul branch corrente vengono ripristinati i dati dell'ultimo commit, sovrascrivendo le eventuali modifiche apportate

**git checkout -b develop** crea il ramo **develop** e lo rende attivo. E' in pratica l'unione dei due comandi precedenti.

**git checkout -m [oldBranchName] newBranchName** rinomina il branch indicato con il nuovo nome. Se non si specifica il nome del branch da rinominare, il nuovo nome viene assegnato al branch corrente.



## Push del branch sul server

---

`git push origin develop` esegue il push del branch corrente sul `develop` di "origin".

`git pull origin develop` esegue il download di `develop` dal server "origin".

`git push --set -upsteam origin develop` memorizza le informazioni di push evitando di doverle ripetere nei push successivi (un po' come per l'opzione `-u`), in cui sarà sufficiente scrivere `git push`

## Il comando git merge

---

I branch di solito sono aperti per interventi di manutenzione e devono essere molto brevi. La regola è quella di creare tanti branch tutti molto brevi. Più branch possono essere aperti in parallelo per eseguire interventi differenti. La cosa importante è che gli interventi siano **isolati**, cioè non interferiscano a vicenda.

Terminato l'intervento su un branch, occorre eseguire il **commit** relativamente a quel branch. Dopo il commit occorre eseguire un **merge** tra la versione appena committata e la versione più recente del master, che potrebbe essere quella da cui siamo partiti o potrebbe invece essere stata modificata a seguito del merge di un altro ramo parallelo.

Il merge riunisce due branch in modo direzionale, nel senso che c'è un branch principale che deve acquisire le modifiche realizzate sull'altro branch di sviluppo. Per eseguire il merge occorre **posizionarsi sul branch "principale" (normalmente il master)** ed utilizzare il comando merge indicando come parametro il nome del branch secondario che deve subentrare:

```
git checkout master      // ritorno sul master
git merge develop        // porta su master gli sviluppi di del ramo develop
```

Nel caso in cui qualcun altro nel frattempo abbia apportato modifiche al master, in corrispondenza del merge si possono generare dei conflitti con fallimento del merge e relativa indicazione dei files che sono andati in conflitto.

In questo caso occorre risolvere manualmente i conflitti e poi rifare il merge.

Con il merge vengono riportati su master tutti i commit via via eseguiti sul branch develop per cui, dopo il merge, si può definitivamente rimuovere il branch develop.

## Visualizzazione dei branch

---

`git branch` elenco completo di tutti i branch in corso di esistenza

`git branch -v` elenco completo di tutti i commit effettuati su tutti i branch esistenti

`git branch --merged` elenco dei branch che hanno avuto merge (compreso master)

`git branch -no-merged` elenco dei branch ancora aperti (non ancora fusi)



## Cancellazione dei branch

---

Una volta che un branch è stato fuso con un altro, la sua esistenza diventa inutile e lo si potrà (eventualmente) eliminare definitivamente mediante l'istruzione

```
git branch -d develop
```

Viceversa i rami che non sono ancora stati fusi con altri non possono essere cancellati con il comando precedenti. Se si è sicuri di voler rimuovere un branch non ancora fuso occorre utilizzare l'opzione **-D**

## Esempio

---

Realizzazione di un remote merge con un branch2 implementato localmente

```
// seleziono il master in locale
git checkout master
// aggiorno il master locale al contenuto di quello remoto
git pull origin master
// seleziono branch2 in locale
git checkout branch2
// copio master all'interno di branch2  (*)
git merge master
// push di branch2 su branch2 del server remoto
git push origin branch2

// seleziono il master in locale
git checkout master
// merge di branch2 all'interno del master
git merge branch2
// upload di master
git push origin master
```

(\*) **Notare il doppio merge.** Il contenuto di master viene "incollato" all'interno di branch2 il quale rimarrà per sempre nella situazione attuale, anche in caso di ulteriori modifiche sul master.

Incollando branch2 dentro master, le successive modifiche di master si ripercuoteranno sulla versione attuale costituita da master + branch2.

## Tag e Numeri di versione

I tag consentono di assegnare un numero di versione ad un **commit** funzionante, stabile, di cui si vuole tenere traccia anche a seguito di modifiche successive. In genere si assegna un tag di versione in corrispondenza di ogni modifica sul master.

Per l'impostazione dei numeri versione si seguono normalmente le regole del **semantic versioning**, cioè

**Major.Minor.Patch**  
**2.3.24**

MAJOR: versione con cambiamenti incompatibili (breaking change)

MINOR: versione con nuove funzionalità, retrocompatibile

PATCH: versione con bug fixes e func fixes

1.0.0 di solito è la prima versione matura, non la prima rilasciata

Ogni puntino dell'alberatura corrisponde ad un COMMIT. Il tag viene visualizzato sull'alberatura a fianco del commit corrispondente. Git Hub visualizza su schede diverse le versione taggate, che dunque possono essere visualizzate e scaricate in modo estremamente veloce.

## Sintassi per la creazione dei tag

L'opzione **-a** consente di definire il nome del tag

L'opzione **-m** definisce un messaggio di commento al tag

```
git tag -a v1.5.2 -m "updating version 1.5.2 with some chores"
```

L'opzione **-d** consente di cancellare un tag

```
git tag -d v1.5.2
```

**git tag** visualizza l'elenco completo di tutti i tag impostati

```
git tag -l 3* visualizza l'elenco dei soli tag del ramo "3.x" (regex allowed)
```

## push del tag

Dopo aver eseguito un commit ed avergli assegnato un tag, occorre eseguire manualmente il **push** del tag tramite uno dei seguenti comandi:

```
git push origin -tags // tutti i tags
git push origin v1.5.2 // il solo tag v1.5.2
```

## Alias dei comandi

Lo scopo degli alias è quello di abbreviare i tempi necessari per la digitazione dei comandi. E' possibile associare un alias a tutti i comandi utilizzati più frequentemente scegliendo come personalizzarli a seconda delle proprie esigenze.

```
git config --global alias.st status // Definisce un alias st per il comando status
git config --global --unset alias.st // Rimuove l'alias st
```

## Il comando git stash

Il comando `git stash` consente di eliminare delle modifiche errate per tornare all'ultima versione committata. Taglia tutte le modifiche non staged del progetto corrente e le salva nello stash.

### `git stash apply`

riapplica le modifiche dallo stash al progetto corrente.

## Cenni sul comando git rebase

Il rebasing è una operazione simile al merging, consente cioè di includere le modifiche eseguite a carico di un ramo in un altro ramo.

Il rebasing è comodo soprattutto nel momento in cui il merge tra due branch genera un conflitto perché nel frattempo qualcun altro ha apportato su uno dei due branch delle modifiche conflittuali.

Il comando

### `git rebase master`

eseguito dal branch in corso di elaborazione, prende tutti i cambiamenti di master e li porta dentro `develop`. Inoltre tutti i nuovi commit del master verranno inclusi anche in `develop`.

Si dice che rebase sposta il ramo "`develop`" all'estremità del master. In pratica *rebase* si occupa di **aggiornare automaticamente la cronologia del progetto**, generando dei nuovi commit su `develop` per ciascun commit effettuato sul ramo rifondato (master).

Il *rebasing* presenta comunque delle controindicazioni che lo rendono uno strumento da adottare con estrema cautela. Il problema principale è che **ha effetto soltanto sul repository di chi lo ha effettuato**. Ipotizziamo di effettuare l'operazione inversa rispetto alla precedente, cioè spostarci sul master ed eseguire

### `git rebase newbranch`

In questo caso l'effetto del *rebasing* è lo spostamento dei commit del master all'estremità di "`develop`". In questo modo ogni commit su `develop` si rifletterà anche su master, per cui **la cronologia del master su cui stiamo lavorando non sarà più la stessa disponibile per gli altri componenti del team**, che hanno invece come riferimento il master originale. Il master avrà quindi due cronologie diverse, una per chi ha eseguito il *rebasing* e l'altra per gli altri sviluppatori.

Se si ha anche il solo sospetto che un altro componente del team stia lavorando allo stesso ramo è sempre opportuno evitare il *rebasing*. Se per un errore di valutazione il *rebasing* ha portato alla duplicazione del master, la soluzione più adeguata per evitare ulteriori danni è quella di sottoporre i due master risultanti ad un *merging*.

## GitHub

**GitHub** è una implementazione cloud di git (altra piattaforma famosa bitbucket). Dal 2018 consente anche di gestire i repository in locale.

I progetti pubblicati su GitHub possono essere **pubblici** o **privati**, cioè visibili soltanto al proprietario e agli stakeholders. Solitamente non è possibile modificare un repository su github, ma è possibile clonare in locale qualsiasi progetto pubblico presente ed eventualmente modificarlo in locale.

Ci si deve innanzitutto registrare inserendo:

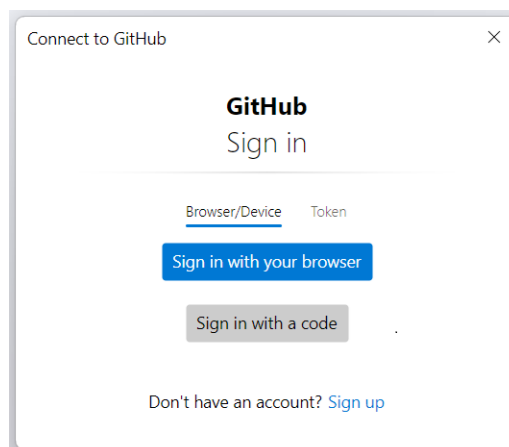
- username
- mail
- password

### Modalità di autenticazione da remoto

La vecchia modalità di login tramite finestra di autenticazione è stata dismessa a partire da gennaio **2021** e sostituita con la creazione di un token che funge sostanzialmente da password per gli accessi da remoto e vale per tutti i repository presenti sul pc.

Nel **2024** anche questa modalità è stata superata da una nuova modalità gestita tramite il cosiddetto Git Credential Manager.

Nel momento in cui si esegue un **git clone** da un repository privato oppure si esegue una prima **git push** su un repository creato completamente vuoto, automaticamente git apre la seguente finestra di autenticazione

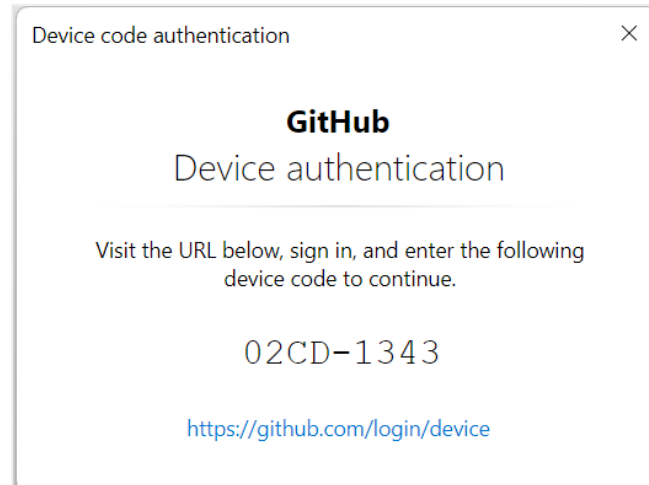


in cui l'utente può scegliere fra tre diverse modalità di autenticazione:

1) **Sign in with your browser** che va a 'cercare' le credenziali di accesso all'interno del browser. Se all'interno del browser è aperta una sessione di github ancora valida, il processo di autenticazione viene gestito in modo completamente trasparente dal *Git Credential Manager* e va a buon fine senza ulteriori interventi.

Se invece all'interno del browser non è presente una sessione di github valida, viene aperta una finestra in cui vengono richiesti username e password di github come se si stesse aprendo una nuova sessione su github

- 2) **Sign in with a code** In corrispondenza di questa scelta viene aperta la seguente finestra che invita l'utente ad eseguire un login su github ed inserire il device code indicato. In questo modo il device rimane abilitato in modo perenne ed eseguire operazioni di push e pull su git hub

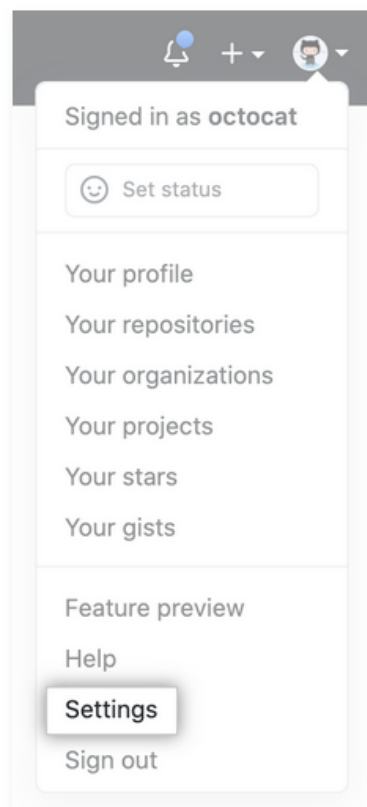


- 3) **Token** compare una finestra di inserimento del Personal Token creato su github.

### Procedura di generazione di un Personal Token

Per la generazione del token occorre andare su github e seguire la seguente procedura:

- 1 In the upper-right corner of any page, click your profile photo, then click **Settings**.



Selezionare quindi

- l'ultima voce in basso **Developer settings**
- **Personal Access Token**
- **Tokens**
- pulsante **Generate New Token** (classic)
- assegnare al token un nome descrittivo : my personal token
- assegnare una data di scadenza
- selezionare i privilegi che si intendono assegnare a questo token.  
E' possibile una gestione dei privilegi abbastanza fine. Ad esempio è possibile creare un token valido per un singolo repository, in modo che chi lo utilizzerà possa accedere soltanto a quel singolo repository.  
Un token senza privilegi può accedere solo alle informazioni pubbliche.  
Per utilizzare il token per accedere ai repository dalla riga di comando, seleziona repo
- Al termine utilizzare il pulsante **Generate Token**

Un apposito messaggio visualizza il token generato raccomandando di salvarlo correttamente perché NON sarà più visibile in futuro.

---

### Salvataggio del token sul PC remoto

---

In corrispondenza di un primo push/clone su un repository privato, git chiede di inserire manualmente il token di accesso. Il token, se valido, viene quindi salvato fra le variabili di sistema nel profilo utente del PC locale e verrà utilizzato automaticamente per tutte le operazioni successive.

I cloni da repository pubblici possono invece essere fatte liberamente senza necessità del token.

---

### Modifica delle credenziali all'interno delle variabili di sistema

---

Nel momento in cui token dovesse scadere, o per qualche ragione essere rigenerato, oppure semplicemente perché si dispone di due account github diversi da utilizzare in situazioni diverse, occorre andare nel Pannello di Controllo di Windows /

#### **Account utente / Gestisci le Credenziali / Credenziali Windows / gitHub**

ed aggiornare la password (cioè il token)

Nel caso in cui si desideri modificare anche lo username, occorre rimuovere fisicamente tutte le precedenti credenziali associate a git / github.

In questo modo, in corrispondenza della prossima push git non trovando credenziali associate, provvederà di nuovo a richiedere le credenziali di accesso a github e a memorizzarle all'interno del Registry di Windows.

## Creazione e Gestione di un Repository su github

Per eseguire il push di un progetto su github occorre prima creare un nuovo repository mediante il pulsante **NEW** in alto a destra.

**Create a new repository**  
A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

**Repository template**  
Start your repository with a template repository's contents.  
No template ▾

**Owner \*** / **Repository name \***  
profMana / mailProject ✓

Great repository names [mailProject is available.](#) [le. Need inspiration? How about curly-pancake?](#)

**Description (optional)**

☒ **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**  
You choose who can see and commit to this repository.

**Initialize this repository with:**  
Skip this step if you're importing an existing repository.

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more.](#)

☐ **Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**  
A license tells others what they can and can't do with your code. [Learn more.](#)

**Create repository**

In fase di creazione di un nuovo repository, Github chiede se deve creare in automatico i files .gitignore e readme.md. Questa scelta dipende da che cosa si vuole fare:

- Se si intende creare un progetto ex-novo che poi scaricheremo in locale conviene che Github crei da subito questi file che verranno poi scaricati in locale in corrispondenza del clone. Questo strada è più rapida in quanto, sul PC locale, sarà sufficiente eseguire un unico comando `git clone`

In questo caso l'opzione **Add .gitignore** è decisamente comodo perchè consente di creare il file .gitignore sulla base del modello indicato. Ad es se si sta sviluppando un progetto Android, verranno automaticamente inserite dentro .gitignore tutte le cartelle ed estensioni da non sottoporre a versioning. E' possibile reperire in rete files .gitignore molto dettagliati relativamente a qualsiasi ambiente di lavoro.

- **Se invece si intende utilizzare il nuovo repository per uploadare in rete un progetto già esistente in locale**, occorre creare un repository completamente vuoto (quindi senza Read.me e gitignore). In caso contrario (cioè se si creano nuovi files su github), in corrispondenza della prima push, sulla destinazione, verrà creato e un nuovo branch !! . Se lo si crea vuoto, subito dopo la creazione vengono visualizzati i comandi da applicare in locale per eseguire il primo commit

## Push del progetto

Al termine della creazione del repository su github, viene visualizzato l'indirizzo di accesso al repository medesimo: `https://github.com/profMana/mailproject.git`

Dal progetto locale (che dovrà già contenere i files `readme.md` e `.gitignore`) lanciare i seguenti comandi:

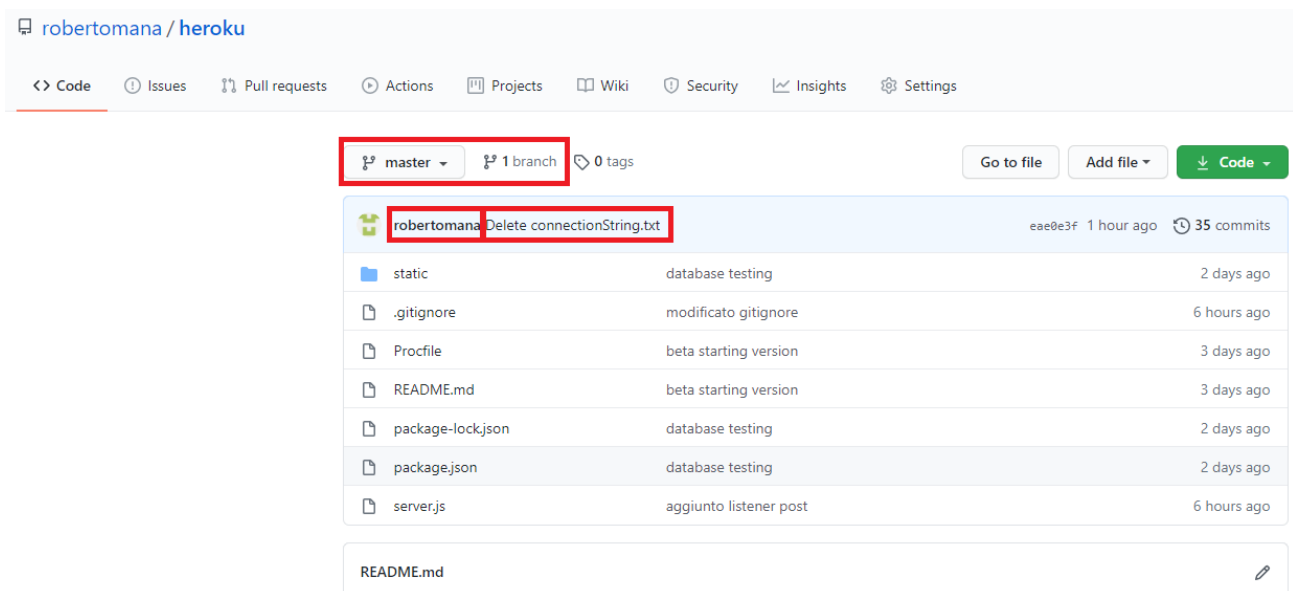
```
git init -b main
git add .
git commit -a -m "first commit"
// git branch -M main
git remote add origin https://github.com/profMana/mailproject.git
git push -u origin main
```

Per push successive, sarà sufficiente fare:

```
git add .
git commit -a -m "new commit"
git push
```

## Navigazione del Repository

Ogni utente può ovviamente gestire più repository. Selezionato un repository, in alto a sinistra compare il nome del repository (ad es nella figura: `robertomana/heroku`)



Al centro il nome del branch correntemente visualizzato (**master**) ed il numero di branch che compongono il repository (nell'esempio **1 solo branch**)

Cliccando a destra sul pulsante verde **Code** si ha la possibilità di :

- clonare l'intero repository, però solo tramite l'applicazione **githubDesktop** che deve essere preventivamente installata sul pc locale (in alternativa è possibile clonare il repository utilizzando il comando `git clone` da terminale)
- Scaricare uno zip della sola ultima versione







Nella finestra di lavoro principale sono inoltre visualizzati:

- il **nome dell'utente** (robertomana)
- il **nome dell'ultimo commit** effettuato ("Delete connectionString.txt")
- Il **contenuto della working directory** al momento del commit

## Visualizzazione dei commit

---

Cliccando sul pulsante **commit** di destra si apre l'elenco di tutti commit effettuati ciascuno con il proprio **UID**:

Update dispatcher.ts ... profMana committed 1 hour ago	Verified		d6392d4	<>
second commit profMana committed 1 hour ago			4334141	<>
Update dispatcher.ts profMana committed 1 hour ago	Verified		7cbf75c	<>
first commit profMana committed 2 hours ago			bb67658	<>

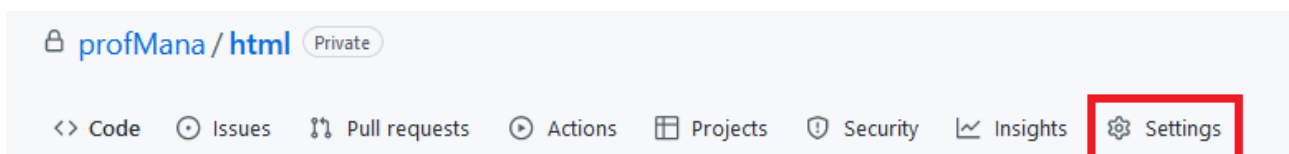
- Cliccando sul nome di un singolo commit oppure sul penultimo pulsante (contenente l'**UID** di quel commit) si aprono i **Dettagli** di quel commit con evidenziate in verde/rosso le modifiche.  
Il comando **Browse File** consente di visualizzare nella finestra principale la **working directory del commit selezionato**. A destra del nome utente e del commit description viene infatti visualizzato l'**UID** del commit selezionato
- L'ultimo pulsante **<>** consente di visualizzare direttamente i files del commit selezionato, senza dover passare attraverso la finestra dei dettagli

I file su github possono anche essere modificati online e quindi committati.

Il client potrà eseguire una pull per scaricare il commit più recente presente sul server ed eventualmente eseguire un merge con il proprio lavoro.

## Impostazioni del repository

Dalla Home page del repository aprire il menù **Settings** (visibile solo DOPO aver fatto il login) che consente la configurazione del singolo repository.



Tra le altre cose, consente di:

- **Rinominare il repository**
- **Aggiungere dei collaboratori** (fra gli utenti registrati di github). Su un repository privato si può inserire un max di 3 collaboratori. I collaboratori riceveranno un messaggio su git di invito a collaborare al progetto.
- **Inviare delle notifiche** a tutti i collaboratori o anche a persone esterne rispetto a github
- La voce iniziale **General** presenta una lunga lista di opzioni / personalizzazioni, fra cui quella di **cancellare l'intero repository**. A tal fine scorrere a fondo pagina fino alla **Danger Zone**

#### Danger Zone

The screenshot shows the 'Danger Zone' section of a GitHub repository settings page. It contains four distinct actions, each with a description and a corresponding button:

- Change repository visibility:** 'This repository is currently private.' with a 'Change visibility' button.
- Transfer ownership:** 'Transfer this repository to another user or to an organization where you have the ability to create repositories.' with a 'Transfer' button.
- Archive this repository:** 'Mark this repository as archived and read-only.' with an 'Archive this repository' button.
- Delete this repository:** 'Once you delete a repository, there is no going back. Please be certain.' with a 'Delete this repository' button.

## Pull Request

Sono il cuore di GIT. Terminato il fix su un certo branch, prima di eseguire il merge con il master, si può inviare una Pull Request a tutti i collaboratori (in particolare il capo progetto) chiedendo di valutare il lavoro fatto sul branch. Se è tutto ok il capo progetto eseguirà la pubblicazione su master. Ogni componente può valutare le modifiche ed esprimere eventuali considerazioni. In questo modo, tra l'altro, tutti i componenti del team saranno portati a programmare più o meno nello stesso modo. Ad esempio se continuo a mettere la graffe per una singola riga di una IF e tutti mi dicono che non va bene, dopo un pò non lo faccio più.

Per lanciare una Pull Request su github esiste un apposito pulsante **New Pull Request** da eseguirsi sul branch di sviluppo appena terminato.

All'interno della Pull Request vengono visualizzate tutte le righe modificate e si potrà valutare come le modifiche andranno ad impattare sul codice del branch di destinazione. Al termine si potrà completare la richiesta cliccando su **"create pull request"** indicando i componenti del gruppo che dovranno dare la loro approvazione.

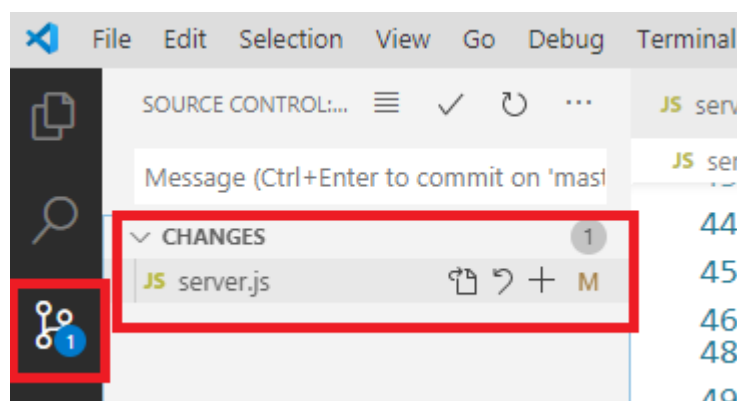
Se le modifiche non vanno in conflitto con quelle degli altri contributori, il sistema mostrerà un alert con cui avvisa che si può effettuare il merge senza rischio di conflitti (i quali, nel caso, dovranno essere risolti manualmente).


## Utilizzo di git all'interno di Visual Studio Code

Visual studio code si integra molto facilmente con git senza necessità di installare nessun plugin. Visual Studio Code va a leggersi direttamente il contenuto del file .git ed agisce di conseguenza.

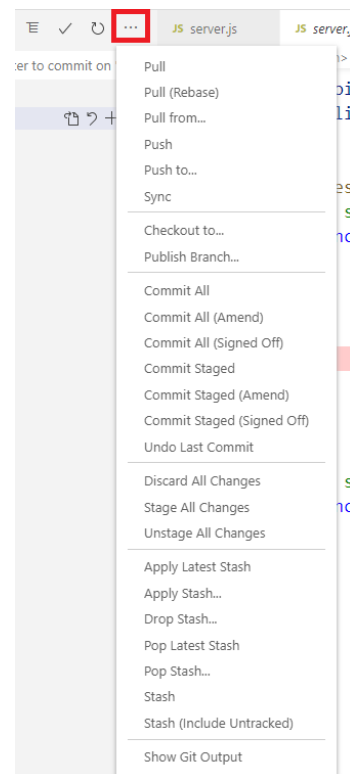
Il 3° pulsante della toolbar (source control) consente di visualizzare le modifiche fatte dopo l'ultimo commit. I 4 pulsantini a fianco di ciascun filename consentono di:

- Aprire l'ultima versione del file
- Annullare le modifiche (git restore)
- Confermare le modifiche salvandole nello stage (git add)
- Confrontare le modifiche rispetto all'ultima versione committata (git diff)



Tutti gli altri comandi git sono disponibili cliccando sui  della barra superiore.

Dal terminale di Visual Studio Code è comunque possibile lanciare manualmente tutti i comandi git



## Prontuario Veloce

---

### Pubblicazione su github

---

- creare in locale una nuova cartella indipendente con l'applicazione posizionata nella root (senza sottocartelle intermedie)
- creare su github un nuovo repository **completamente vuoto**, senza README e senza GITIGNORE che dovranno invece essere creati manualmente nel repository locale da uploadare
- nella root della cartella locale lanciare i seguenti comandi:

```
git init -b main
git add .
git commit -a -m "first commit"
// git branch -M main
git remote add origin https://github.com/profMana/crudServer.git
git push -u origin main
```

Per push successive, sarà sufficiente fare:

```
git add .
git commit -a -m "new commit"
git push
```

### Download su un nuovo PC

---

- da qualsiasi posizione (di solito direttamente sul **desktop**, in quanto poi github restituisce la cartella principale del progetto con tutto il suo contenuto) lanciare il comando:

```
git clone https://github.com/profMana/crudServer.git
git clone https://github.com/vallauri-ict/tpsi-playground-profMana-3B.git
```

il quale, nella posizione di lancio, crea una **sottocartella** GIT avente il nome del progetto clonato (**crudServer** oppure **tpsi-palyground-profMana**) e posiziona al suo interno tutto il contenuto clonato e anche l'apposita cartella .git

### Download su un PC già contenente il progetto

---

```
// git remote add origin https://github.com/profMana/crudServer.git
git pull origin main    (origin e branch devono essere sempre indicati)
```