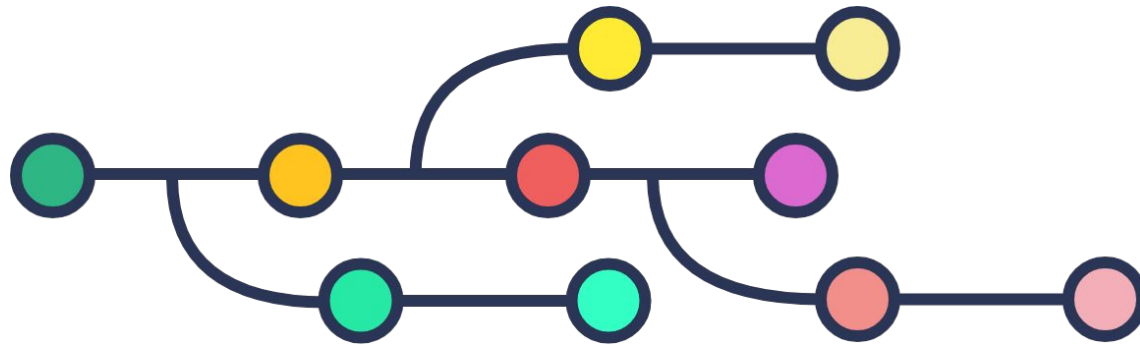


# Controllo di versione



Montanari Alessandro  
A.S. 2020/2021

# Controllo di versione

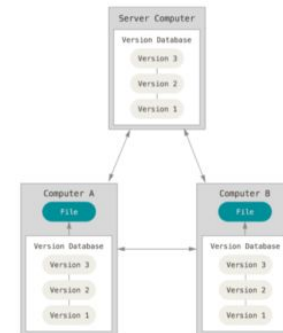
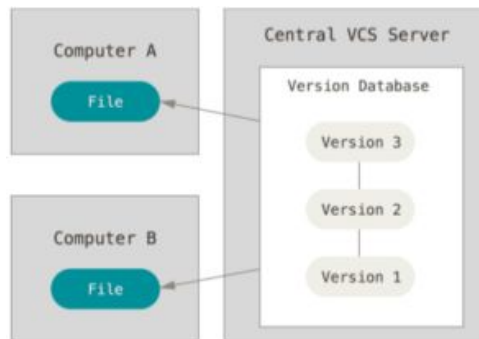
Il controllo di versione (versioning) è un meccanismo attraverso il quale si tiene traccia, con il passare del tempo, delle modifiche apportate ad un progetto (composto solitamente da più file).

Nello specifico dello sviluppo software, l'impiego di un sistema per il controllo di versione consente di semplificare il lavoro di un team, fornendo ai suoi componenti gli strumenti per monitorare i cambiamenti effettuati da terzi ad ogni revisione, per risalire ad un intervento che potrebbe aver causato un malfunzionamento, per annullare una variazione ripristinando la precedente release di un file o di un progetto nel suo insieme.

Il sistema che offre questi strumenti si chiama Version Control System (VCS)

# Tipologie di VCS

- Sistema di Controllo di versione locale (VCS) → gestiti localmente sulla singola macchina
- Sistema di Controllo di versione centralizzato (CVCS) → gestiti in remoto su un server centralizzato
- Sistema di Controllo di versione distribuito (DVCS) → la copia principale è gestita da un server in remoto, ma ogni client possiede tutta la storia del progetto



# Controllo di versione locale (manuale)

Si immagini uno sviluppatore che crea un'applicazione e poi, per esigenze di varie natura, 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. Teoricamente tale approccio non necessita di un software per la gestione delle diverse release. Ma cosa accadrebbe nel caso di un progetto più articolato nel quale sono coinvolte numerose modifiche?



Progetto  
2020\_03\_07  
11.00



Progetto  
2020\_03\_09  
08.45



Progetto  
2020\_03\_09  
10.50



Progetto  
2020\_03\_10  
12.30



Progetto  
2020\_03\_10  
17.33



Progetto  
2020\_03\_10  
21.00

# Controllo di versione locale (manuale)

Ma cosa accadrebbe nel caso di un progetto più articolato nel quale sono coinvolte numerose modifiche?

- Tenere traccia degli interventi effettuati diventerebbe complesso
- Rischio di perdere informazioni con il passare del tempo
- Accumulazione degli interventi eseguiti



Progetto  
2020\_03\_07  
11.00



Progetto  
2020\_03\_09  
08.45



Progetto  
2020\_03\_09  
10.50



Progetto  
2020\_03\_10  
12.30



Progetto  
2020\_03\_10  
17.33



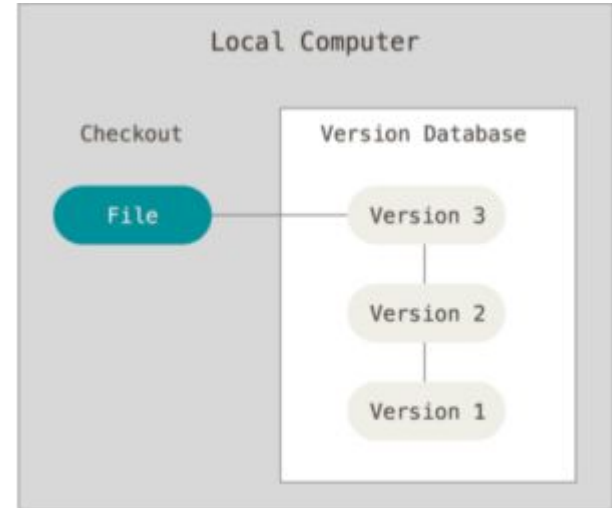
Progetto  
2020\_03\_10  
21.00

# Controllo di versione locale

Un sistema LVCS (Local Version Control System) si basa su un unico database che gestisce tutti i cambiamenti subiti dai file di cui si vuole tenere traccia della versione.

Un sistema di versioning locale risiede su una macchina e con l'ausilio di un database gestisce le versioni dei file di un progetto.

Un tale sistema è, però, poco adatto per quelle situazioni in cui più persone partecipano al progetto e quindi hanno la necessità di condividere e modificare i file.

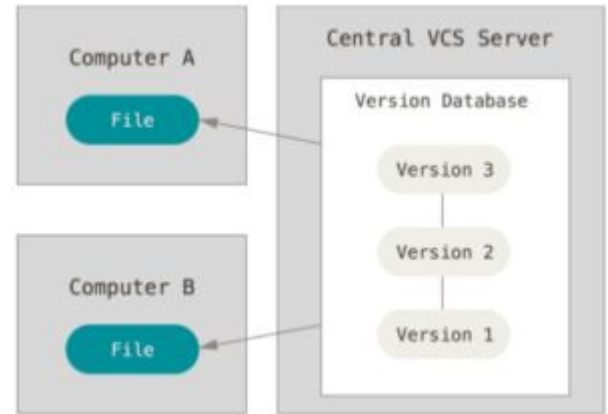


# Controllo di versione centralizzato

Un sistema VCS (Version Control System) si basa su un unico nodo (Central VCS server) che gestisce il database delle versioni. I client si collegano al sistema, scaricano i file, apportano le modifiche e alla fine aggiornano la situazione sul nodo centrale.

Il vantaggio di questo sistema è che in ogni momento tutti hanno la piena conoscenza dello stato del progetto e di cosa ogni partecipante al progetto stia facendo.

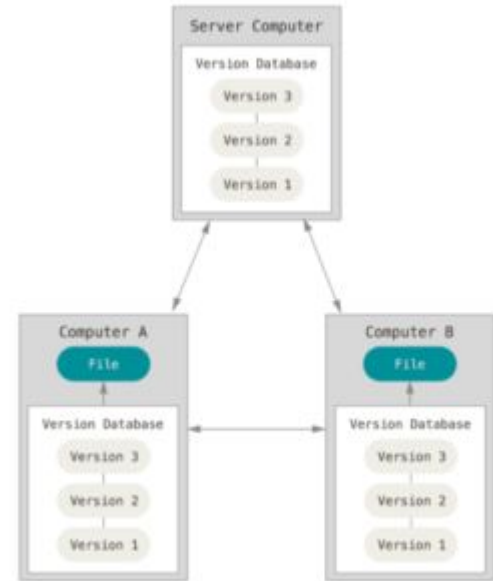
Svantaggio: essendoci un solo nodo che gestisce il tutto, qualora il server centrale non fosse accessibile, nessuno sarebbe in grado di lavorare.



# Controllo di versione distribuito

Un sistema DVCS (Distributed Version Control System), si basa su nodo che gestisce il database principale delle versioni. In aggiunta i client possiedono una copia completa della storia del progetto, non hanno bisogno di informazioni provenienti da altri computer, e quando c'è una nuova versione da condividere, il client esegue un'operazione di merge sul repository centrale o su quello di un altro client che partecipa al progetto.

Vantaggio: nel caso di un'interruzione del servizio da parte dei server il repository presente in un client qualunque potrà essere utilizzato per il ripristino.







Montanari Alessandro  
A.S. 2020/2021

# Qualche considerazione

Questo documento non si pone come tutorial completo ed esaustivo sull'argomento Git e GitHub; piuttosto è dedicato ad un primo approccio di tipo scolastico al versioning remoto. La parte teorica, per quanto noiosa, è necessaria per comprendere le potenzialità dello strumento e i relativi comandi.

Esistono molti tool grafici che permettono di risolvere questo tipo di problematiche (sourcetree per esempio), ma non è consigliato il loro utilizzo finché non si ha una buona padronanza dei concetti e dei comandi elencati nel documento.

Di seguito troverete comandi e concetti base, necessari per utilizzare Git e GitHub in maniera più articolata.

Chi utilizza Git? Per esempio [Spotify](#), [Netflix](#), [Apple](#), [Adidas](#), [Youtube](#), [Google](#) e... lo stesso [GitHub](#)!

# Link di riferimento

- [Git - la guida di html.it](#)
- [Git - la guida di Mr. Webmaster](#)
- [Git - la guida tascabile](#)
- [Git cheat sheet](#)
- [GitHub - la guida di vixr](#)
- [GitHub - la guida di GitHub](#)

# Git - qualche informazione

- è uno strumento per lo sviluppo in ambito collaborativo e per l'implementazione di progetti Open Source
- è un **DVCS** (Distributed Version Control System), cioè sistema software per il controllo di versione distribuito
- è stato realizzato nel 2005 da Linus Torvalds, informatico scandinavo a cui si deve anche la paternità del Kernel Linux

# Controllo di versione distribuito - Git

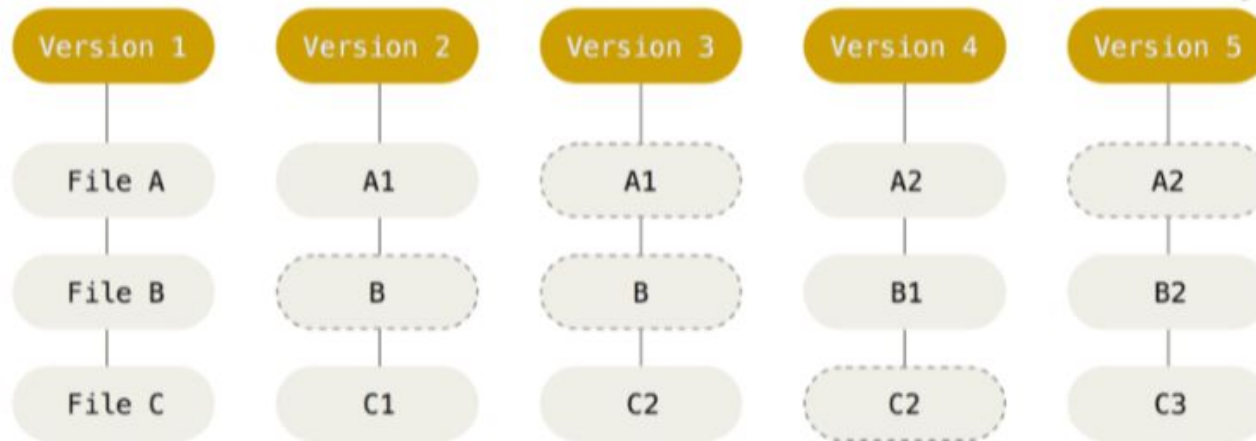
Git rientra nei sistemi DVCS.

I repository sono definibili come degli archivi per la raccolta ordinata dei dati relativi ad un progetto (o più semplicemente, un progetto in Git).

Git (in generale i DVCS):

- i dati vengono rappresentati come delle istantanee (snapshots). Quando si memorizza lo stato di un progetto il sistema crea un'istananea di tutti i file al momento corrente
- non fa riferimento alle differenze tra le varie versioni, ma ne “fotografa” gli stati di avanzamento
- prevede che il controllo degli ultimi snapshot generati venga affidato ai client che eseguiranno delle copie complete dei repository
- permettono di operare su più repository in remoto

# Controllo di versione distribuito - Git



- La figura mostra uno scenario tipico dello stato di avanzamento di un progetto gestito con Git.
- Nella figura i nodi tratteggiati indicano che quel file non ha subito variazioni.

# File in Git

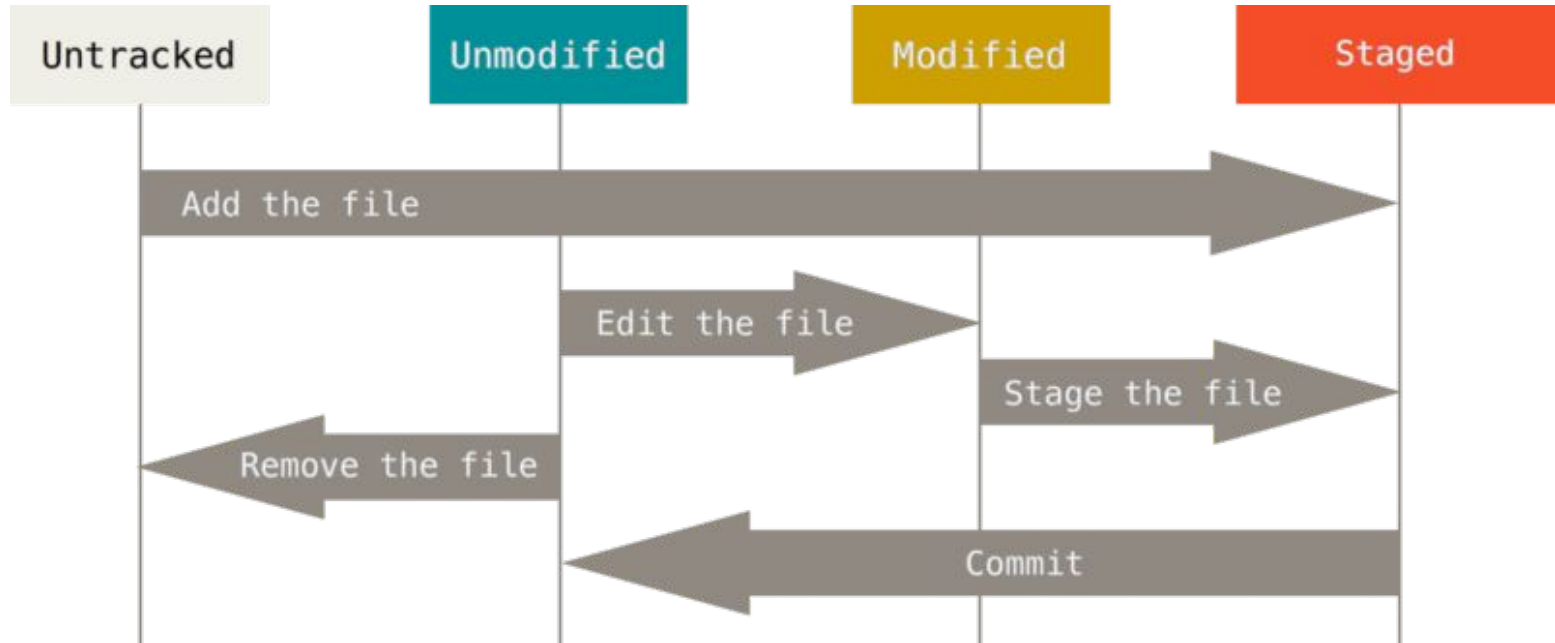
Quando parliamo di Git dobbiamo distinguere tra i file

- Untracked → i file non vengono gestiti da Git
- Tracked → i file vengono gestiti da Git

I file tracked possono assumere stati differenti:

- Committed → i file sono memorizzati localmente nel database di Git
- Modified → i file hanno subito dei cambiamenti ma non sono stati ancora committati
- Staged → i file sono stati coinvolti da modifiche e verranno inclusi in un'istantanea con il prossimo commit (operazione attraverso la quale i file staged vengono memorizzati localmente nel database di Git)
- Unmodified → i file non hanno subito dei cambiamenti dall'ultimo commit

# File in Git







# Test pratico

Montanari Alessandro  
A.S. 2020/2021

# Installare Git

Recarsi al seguente link e scaricare la release a seconda del proprio sistema operativo:

<https://git-scm.com/downloads>

Lasciare le impostazioni di default durante l'installazione. Per verificare la corretta installazione di Git aprire un terminale, come "Git Bash" (installato con il programma precedente). Su windows si può utilizzare anche il "Command Prompt". Digitare il comando `git --version` e premere invio. Dovrebbe comparire una risposta simile: `git version 2.24.1.windows.2`

# Configurare Git

In Git i commit (che in pratica consistono nella memorizzazione delle istantanee di un progetto) fanno riferimento a chi li ha effettuati. Ogni collaboratore dovrà quindi essere dotato di un username e di un indirizzo di posta elettronica che ne definiscono l'identità (da memorizzare in una variabile globale di Git). Di seguito i comandi da lanciare (cioè scrivere su terminale):

- Per memorizzare il proprio username → `git config --global user.name 'tuoUsername'`
- Per memorizzare la propria email → `git config --global user.email laTua@email.com`

Per verificare il corretto salvataggio di queste variabili globali lanciare i seguenti comandi (che restituiranno il valore precedentemente salvato):

- `git config user.name`
- `git config user.email`

# Configurare Git - esempio

```
MINGW64:/c/Users/allen

allen@DESKTOP-VBMRNV8 MINGW64 ~
$ git --version
git version 2.24.1.windows.2

allen@DESKTOP-VBMRNV8 MINGW64 ~
$ git config --global user.name 'Alessandro'

allen@DESKTOP-VBMRNV8 MINGW64 ~
$ git config --global user.email 'emailSuperFasulla@gmail.com'

allen@DESKTOP-VBMRNV8 MINGW64 ~
$ git config user.name
Alessandro

allen@DESKTOP-VBMRNV8 MINGW64 ~
$ git config user.email
emailSuperFasulla@gmail.com

allen@DESKTOP-VBMRNV8 MINGW64 ~
$
```

Controllo la corretta  
installazione

Imposto l'username

Imposto la mail

Controllo l'username

Controllo la mail

# Creare un progetto Git

Creare un nuovo progetto Git significa trasformare la directory contenente il codice in un repository per il versioning. Nel gergo di Git questa operazione prende il nome di import.

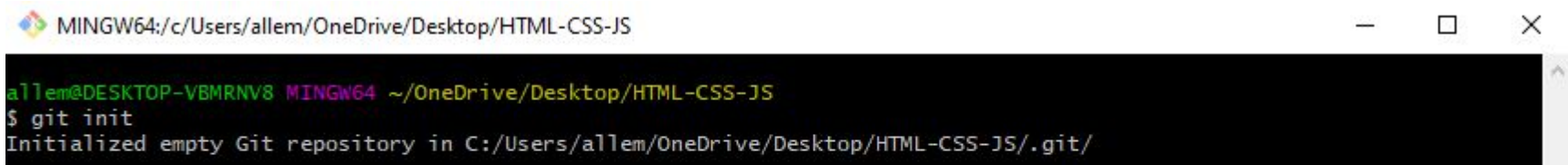
Occorre prima spostarsi (sempre da terminale) all'interno della directory che ci interessa e lanciare il seguente comando:

```
git init
```

Per spostarsi tra le directory con il terminale:

- Tramite comando `cd` (trovi un esempio [qui](#))
- Facendo click destro dentro la directory desiderata e selezionando “Git bash qui”

# Creazione di un progetto



```
MINGW64:/c/Users/allem/OneDrive/Desktop/HTML-CSS-JS
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS
$ git init
Initialized empty Git repository in C:/Users/allem/OneDrive/Desktop/HTML-CSS-JS/.git/
```

Una volta inizializzato un progetto troviamo nella directory utilizzata (nell'esempio in alto la directory è "HTML-CSS-JS") una sottodirectory ".git", impostata come nascosta (se non è stata attivata l'opzione in impostazioni non si vedrà, ma non è un problema).

La directory ".git" contiene i dati dello storico del progetto, i suoi snapshot.

# Primo commit - esempio

A titolo dimostrativo creiamo dentro la cartella “HTML-CSS-JS” due file:

- index.html → nel quale metto un piccolo titolo
- README.md → file senza estensione che uso come file di testo, e scrivo “# Il mio primo README”

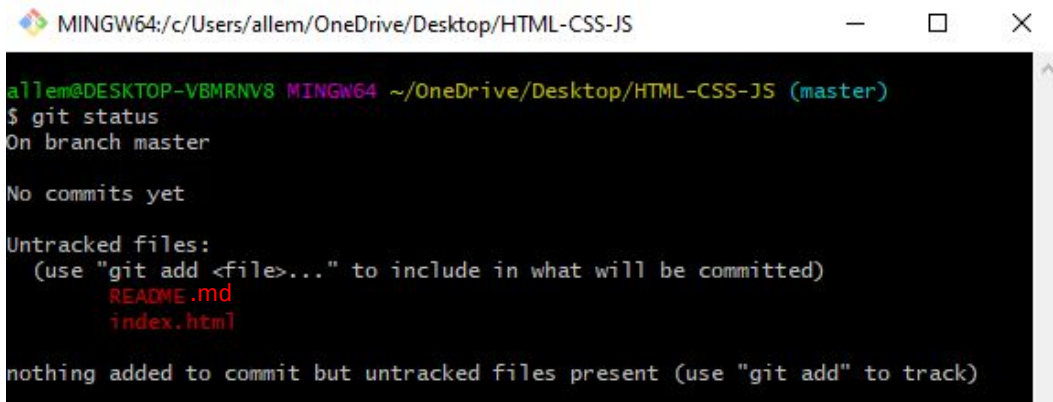


# Primo commit - esempio

Per verificare lo stato dei file posso lanciare il comando (sempre all'interno della directory corretta):

```
git status
```

che mi restituirà un messaggio simile:



```
MINGW64:/c:/Users/allem/OneDrive/Desktop/HTML-CSS-JS
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <File>..." to include in what will be committed)
        README.md
        index.html

nothing added to commit but untracked files present (use "git add" to track)
```



# Primo commit - esempio

Come notiamo i file sono nello stato “untracked”. Per far sì che Git ne tenga traccia si deve lanciare il comando “git add” seguito dal nome del file da aggiungere. In questo caso:

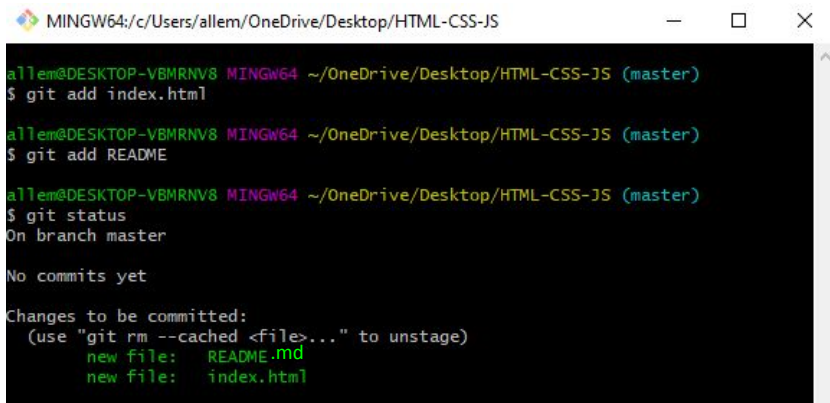
```
git add index.html
```

```
git add README.md
```

Oppure in alternativa, per aggiungere tutti i file nella directory corrente:

```
git add .
```

# Primo commit - esempio



```
MINGW64: c:/Users/allem/OneDrive/Desktop/HTML-CSS-JS
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git add index.html
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git add README
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
    new file:   index.html
```

Lanciando di nuovo il comando “git status” verrà segnalato che i file “README.md” e “index.html” sono passati dallo stato “untracked” allo stato “staged” (e quindi d’ora in poi sono tracked).

NB: se il nome del file è composto da parole separate da spazi, nel momento di aggiungerlo bisogna inserire le virgolette, per esempio:

```
git add “file molto lungo.txt”
```

# Primo commit - esempio

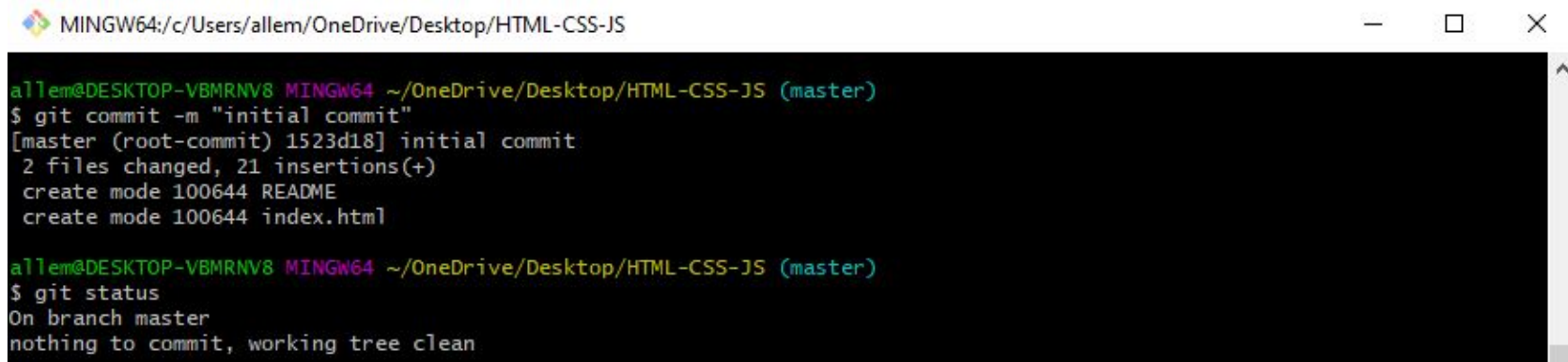
Ora, per fare in modo che tutti i file nello stato “staged” vengano salvati nel database di Git, dobbiamo lanciare il seguente comando (dove al posto di “messaggio” si deve inserire una descrizione di quello che è accaduto dall’ultimo commit):

```
git commit -m “messaggio”
```

Se è il primo commit del progetto, solitamente si usa come descrizione “initial commit”:

```
git commit -m “initial commit”
```

# Primo commit - esempio



```
MINGW64:/c/Users/allem/OneDrive/Desktop/HTML-CSS-JS

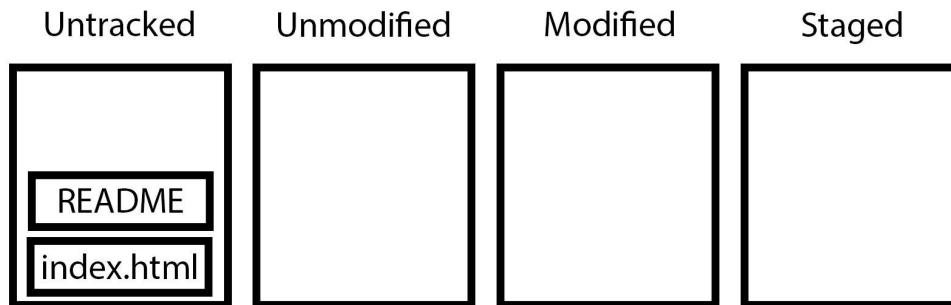
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git commit -m "initial commit"
[master (root-commit) 1523d18] initial commit
 2 files changed, 21 insertions(+)
 create mode 100644 README
 create mode 100644 index.html

allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Verificando ancora lo stato dei file, ci viene detto che non c'è nulla da committare. Vuol dire che tutti i file sono nello stato "unmodified"

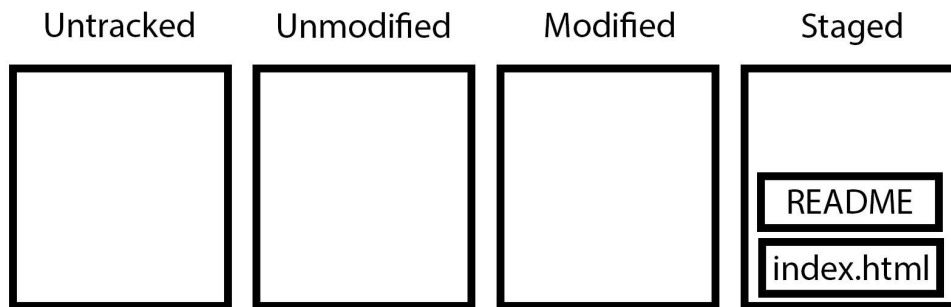
# Primo commit - riassunto

1) Creazione dei file "index.html" e "README.md"



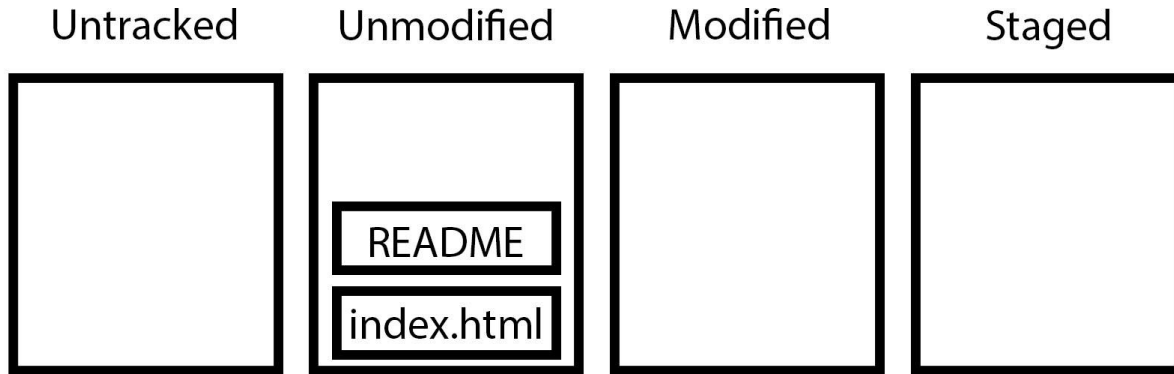
2. Comandi:

- `git add README.md`
- `git add index.html`



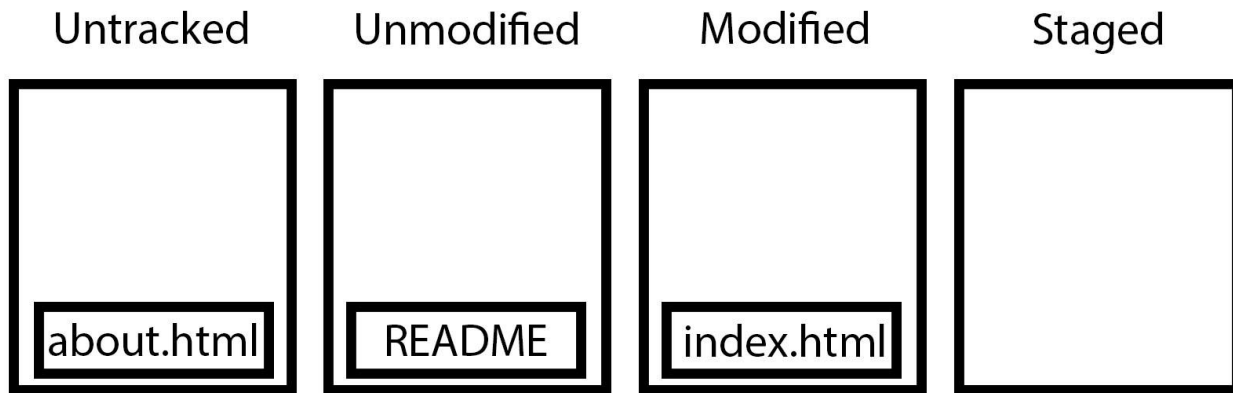
# Primo commit - riassunto

3) Comando: `git commit -m "initial commit"`



## Secondo commit - esempio

Ora aggiungiamo il file “about.html” e modifichiamo il file “index.html”. Ci troveremo nella seguente situazione:

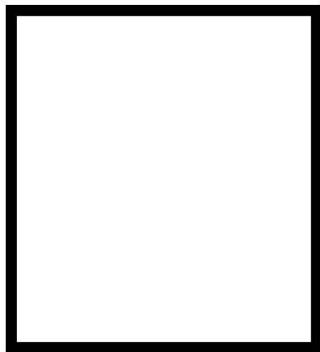


# Secondo commit - esempio

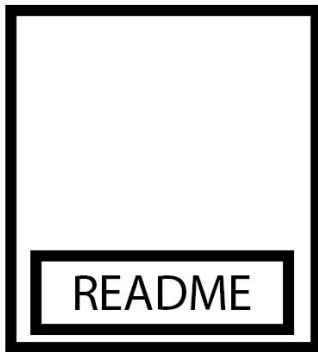
Aggiungiamo entrambi i file nella sezione staged con il comando:

```
git add .
```

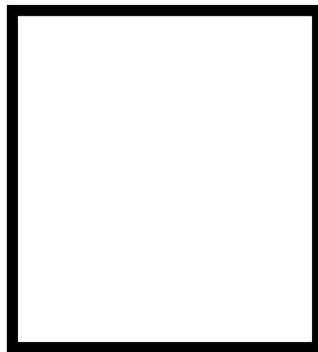
Untracked



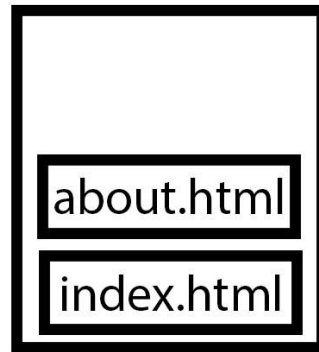
Unmodified



Modified



Staged



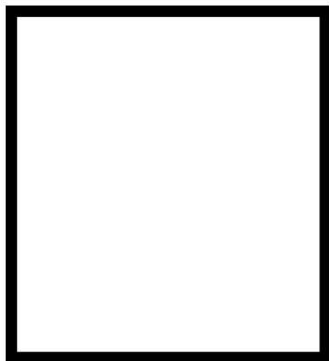


# Secondo commit - esempio

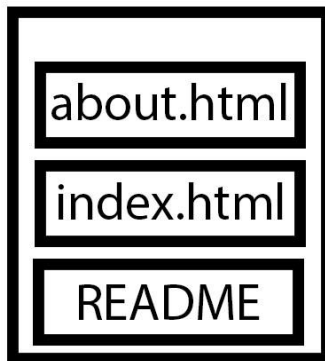
Eseguiamo il secondo commit con il comando:

```
git commit -m "Aggiunto pagina di about e collegata all'index"
```

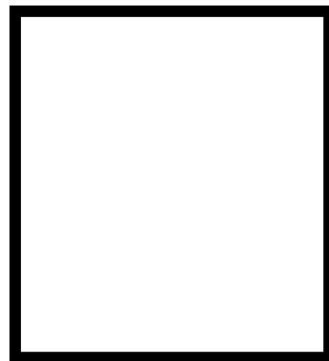
Untracked



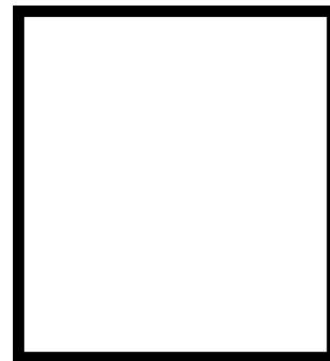
Unmodified



Modified



Staged

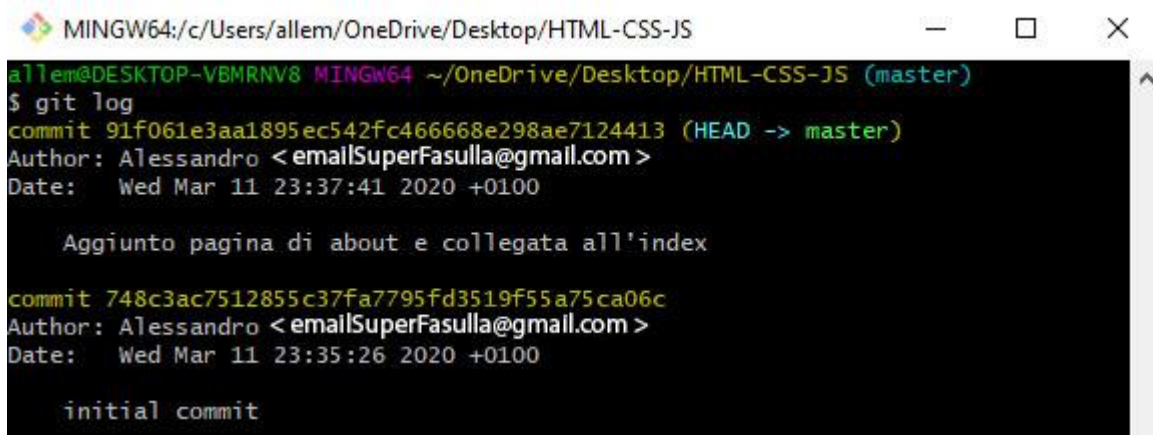


# Storico Git

Per poter ottenere lo storico dei commit eseguiti lanciare il comando:

```
git log
```

Per visualizzare lo storico con un aggiunta grafica aggiungere il flag “--graph”



```
MINGW64; c:/Users/allem/OneDrive/Desktop/HTML-CSS-JS
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git log
commit 91f061e3aa1895ec542fc466668e298ae7124413 (HEAD -> master)
Author: Alessandro <emailSuperFasulla@gmail.com>
Date:   Wed Mar 11 23:37:41 2020 +0100

    Aggiunto pagina di about e collegata all'index

commit 748c3ac7512855c37fa7795fd3519f55a75ca06c
Author: Alessandro <emailSuperFasulla@gmail.com>
Date:   Wed Mar 11 23:35:26 2020 +0100

    initial commit
```

# Storico Git

A fianco di ogni commit è presente una sequenza di caratteri alfanumerici, che rappresentano il suo checksum in SHA-1, detto anche **hash** del commit. È una sorta di identificativo univoco del commit eseguito.

Nell'esempio precedente:

- Primo commit → 91f061e3aa1895ec542fc466668e298ae7124413
- Secondo commit → 748c3ac7512855c37fa7795fd3519f55a75ca06c

# Ritornare ad un particolare commit

Per tornare temporaneamente ad un particolare commit, eseguire il comando:

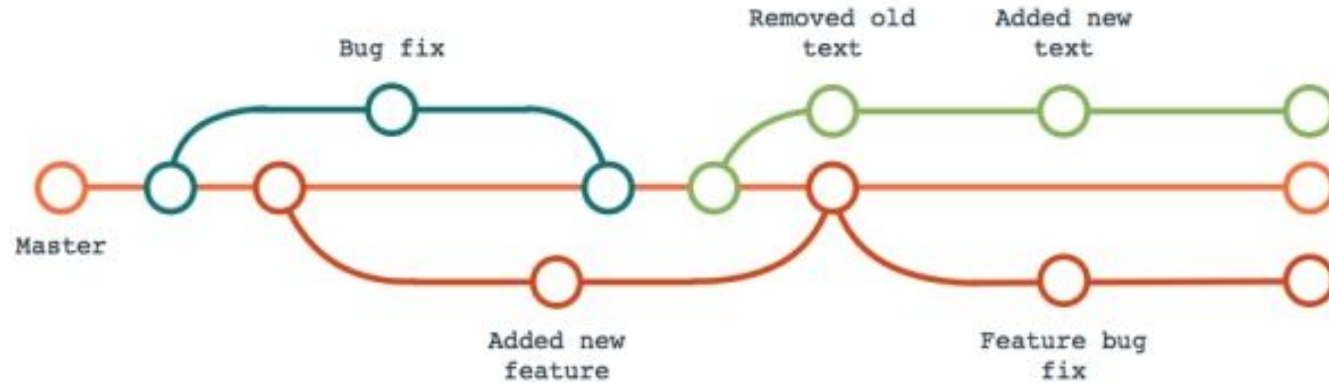
```
git checkout hashDelCommit
```

Se volessi per esempio tornare al primo commit dovrei eseguire il comando:

```
git checkout 91f061e3aa1895ec542fc466668e298ae7124413
```

Per mantenere le modifiche aggiungere “--soft” al comando

# Branch



Montanari Alessandro  
A.S. 2020/2021

# Branch

I branch (ramificazioni) vengono utilizzati in git per l'implementazione di funzionalità tra loro isolate, cioè sviluppate in modo indipendente l'una dall'altra ma a partire dalla medesima base.

Strutturalmente il ramo predefinito di un progetto gestito tramite il DVCS è il master che viene generato quando si crea un repository; sarà poi possibile creare ed utilizzare ulteriori diramazioni dedicate a features differenti per poi inserirle nel master (tramite una procedura detta merging).

Per esempio, le modifiche fatte in un ipotetico branch feature\_1 non intaccano un altro ipotetico branch feature\_2 e il branch principale master, perché si lavora in “spazi virtuali” separati!

# Branch - esempio [ creazione ]

Per creare un nuovo branch per aggiungere la “feature” di un menù dobbiamo eseguire il comando:

```
git branch menuBranch
```

Per modificare il branch corrente (master) e spostarsi sul branch menuBranch dobbiamo eseguire il comando:

```
git checkout menuBranch
```

Ora, guardando i file contenuti all'interno del repository non notiamo alcuna differenza, questo perché questo nuovo branch parte proprio da una copia del branch dal quale si è ramificato, ossia master. Qualsiasi modifica noi facciamo d'ora in poi intaccherà solo ed unicamente il branch menuBranch.

Per verificare il branch corrente: `git branch`

# Branch - esempio [ creazione ]

MINGW64:/c/Users/allem/OneDrive/Desktop/HTML-CSS-JS

```
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git branch menuBranch
```

**Creo il nuovo branch**

```
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git branch
* master
  menuBranch
```

**Controllo il branch corrente  
(master)**

```
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git checkout menuBranch
Switched to branch 'menuBranch'
```

**Mi sposto sul branch corretto**

```
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (menuBranch)
$ git branch
  master
* menuBranch
```

**Controllo il branch corrente  
(menuBranch)**

Da notare che il branch corrente viene visualizzato nella parte destra in azzurro



# Branch - esempio [ utilizzo ]

Ora:

1. Aggiungiamo la feature menù al file "index.html"
2. Aggiungiamo il file all'area di stage
3. Facciamo il commit dei cambiamenti
4. Controlliamo il log dei commit

Se ora torniamo sul branch master tramite il comando

```
git checkout master
```

Il file "index.html" sarà quello vecchio, non quello appena modificato!

# Branch - esempio [ utilizzo ]

```
MINGW64:/c/Users/allem/OneDrive/Desktop/HTML-CSS-JS
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (menuBranch)
$ git add .

allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (menuBranch)
$ git commit -m "Aggiunto feature menu"
[menuBranch 6849aae] Aggiunto feature menu
1 file changed, 3 insertions(+), 1 deletion(-)

allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (menuBranch)
$ git log --graph
* commit 6849aae71a4cb437b1336a45dfe6833e2e7c1e85 (HEAD -> menuBranch)
  Author: Alessandro <emailSuperFasulla@gmail.com>
  Date:   Sun Mar 15 23:56:20 2020 +0100

      Aggiunto feature menu

* commit 91f061e3aa1895ec542fc466668e298ae7124413 (master)
  Author: Alessandro <emailSuperFasulla@gmail.com>
  Date:   Wed Mar 11 23:37:41 2020 +0100

      Aggiunto pagina di about e collegata all'index

* commit 748c3ac7512855c37fa7795fd3519f55a75ca06c
  Author: Alessandro <emailSuperFasulla@gmail.com>
  Date:   Wed Mar 11 23:35:26 2020 +0100

      initial commit
```

# Branch - merging

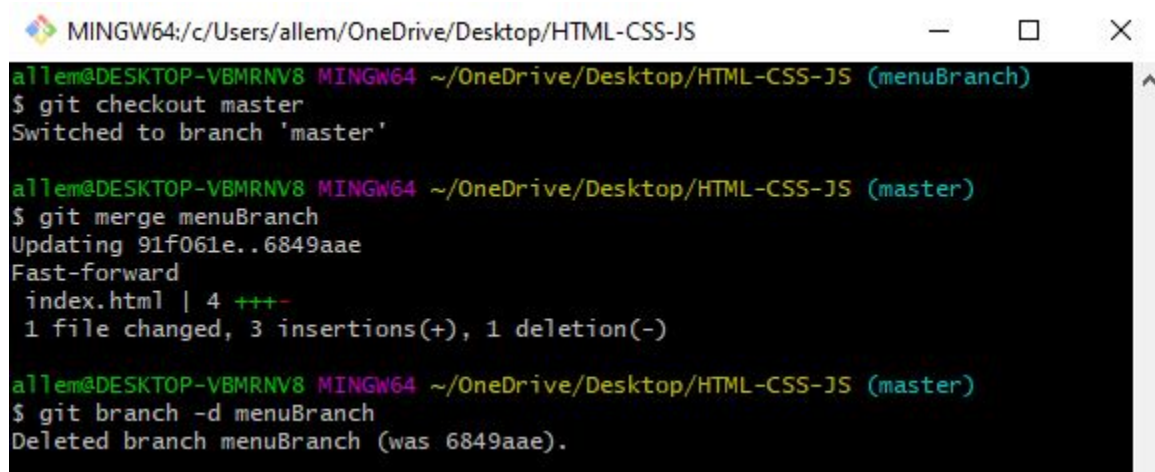
Dopo che la feature è stata completata (oppure una vulnerabilità è stata risolta) non abbiamo più bisogno del branch in questione e possiamo ricorrere al **merging** per fonderlo con il branch master (o al branch dal quale si è ramificato). Per fare il merge ci si deve trovare sul branch di destinazione e si utilizza il comando:

```
git merge nomeBranch
```

Si può poi procedere all'eliminazione del branch non più necessario attraverso il comando:

```
git branch -d nomeBranch
```

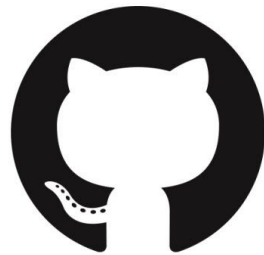
# Branch - esempio [ merging ]



```
MINGW64:/c:/Users/allem/OneDrive/Desktop/HTML-CSS-JS
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (menuBranch)
$ git checkout master
Switched to branch 'master'

allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git merge menuBranch
Updating 91f061e..6849aae
Fast-forward
 index.html | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git branch -d menuBranch
Deleted branch menuBranch (was 6849aae).
```



# GitHub

Montanari Alessandro  
A.S. 2020/2021

# GitHub

Nella sezione precedente abbiamo lavorato solo ed unicamente in locale... e ora? Per condividere il nostro codice e salvarlo online così da averlo a disposizione su più dispositivi, necessitiamo di un servizio esterno: GitHub è il più conosciuto ed utilizzato.

Github è una piattaforma di code sharing, un servizio di hosting per progetti software. Il nome deriva dal fatto che questo è un'implementazione dello strumento di controllo distribuito Git.

Per utilizzare la piattaforma si deve creare un account dal sito:

<https://github.com/>

# GitHub - esempio [ creazione repository ]

Aggiungiamo ora una nuova repository su GitHub nella quale collegheremo la nostra repository locale.

Impostiamo un nome (per esempio test) e aggiungiamo una descrizione.

Possiamo selezionare tra:

- Pubblica → visibile da tutti gli utenti
- Privata → visibile solo dall'utente proprietario (ed eventualmente da altri account scelti)

 allemonta ▼

Repositories



Find a repository...

# GitHub - esempio [ creazione repository ]

## Create a new repository

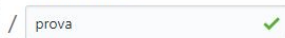
A repository contains all project files, including the revision history. Already have a project repository elsewhere?

[Import a repository.](#)

Owner



Repository name \*



Great repository names are short and memorable. Need inspiration? How about **ideal-guacamole**?

Description (optional)

Questa è una repository di prova



**Public**

Anyone can see this repository. You choose who can commit.



**Private**

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer.

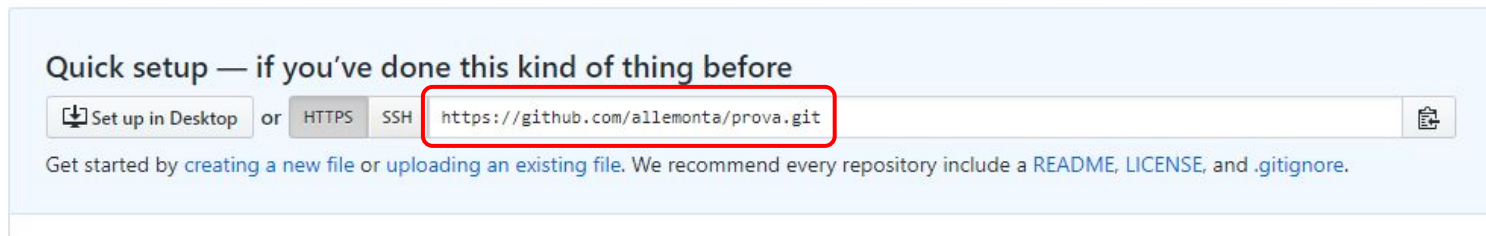
Add .gitignore: **None** ▼

Add a license: **None** ▼ ⓘ

Create repository



# GitHub - esempio [ creazione repository ]



Nella pagina seguente comparirà una sezione contenente un url del tipo:

`https://github.com/username/nomeRepository.git`

Questo è l'indirizzo remoto della nostra repository!

# GitHub - esempio [ collegamento remoto ]

Collegiamo ora il nostro progetto Git locale con il repository remoto con il comando:

```
git remote add origin urlRepositoryRemoto
```

Nel nostro caso:

```
git remote add origin https://github.com/allemonta/prova.git
```

Come anche indicato nella pagina di GitHub:

**...or push an existing repository from the command line**

```
git remote add origin https://github.com/allemonta/prova.git  
git push -u origin master
```

# GitHub - esempio [ sincronizzazione ]

Ora, per inviare lo storico delle modifiche eseguite nel branch master locale al branch origin remoto utilizziamo il comando:

```
git push -u origin master
```

Qui verranno richieste (la prima volta) le credenziali di GitHub.

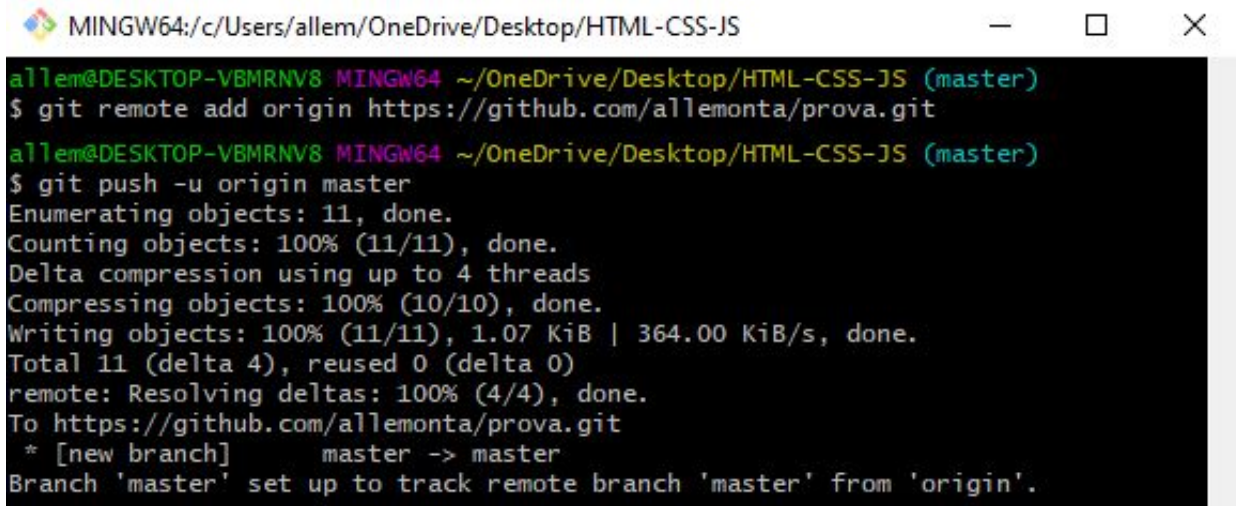
Controlliamo ora la nostra repository remota, navigando sul sito oppure raggiungendola tramite link, la cui struttura è sempre la stessa:

```
https://github.com/username/nomeRepository
```

Nel nostro caso:

```
https://github.com/allemonta/prova
```

# GitHub - esempio [ sincronizzazione ]



```
MINGW64:/c:/Users/allem/OneDrive/Desktop/HTML-CSS-JS
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git remote add origin https://github.com/allemonta/prova.git
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop/HTML-CSS-JS (master)
$ git push -u origin master
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 4 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 1.07 KiB | 364.00 KiB/s, done.
Total 11 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), done.
To https://github.com/allemonta/prova.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

# GitHub - esempio [ controllo ]

[allemona](#) / [prova](#)

Unwatch 1Star 0Fork 0

[Code](#) [Issues 0](#) [Pull requests 0](#) [Actions](#) [Projects 0](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

Questa è una repository di prova [Edit](#)

[Manage topics](#)

3 commits

1 branch

0 packages

0 releases

0 contributors

Branch: master [New pull request](#) [Create new file](#) [Upload files](#) [Find file](#) [Clone or download](#)

**Alessandro** Aggiunto feature menu Latest commit 6849aae 1 hour ago

<a href="#">README.md</a>	initial commit	4 days ago
<a href="#">about.html</a>	Aggiunto pagina di about e collegata all'index	4 days ago
<a href="#">index.html</a>	Aggiunto feature menu	1 hour ago

# GitHub - esempio [ riassunto ]

Fino ad ora abbiamo eseguito i seguenti passaggi:

- Creazione progetto Git in locale → `git init`
- Aggiunta di commit in locale → `git add .` → `git commit -m "commento"`
- Creazione repository su Github
- Collegamento remoto → `git remote add origin urlRepositoryRemoto`
- Aggiunta delle modifiche in remoto → `git push -u origin master`

Ora possiamo modificare il repository locale ed aggiungere altri commit. Per poter sincronizzare queste modifiche con il repository locale usiamo il comando:

```
git push
```

# GitHub - clonazione

Se invece un repository (del proprio account o di quello di un altro) è già stato creato e sono già stati inseriti dei commit, come facciamo ad utilizzarlo? Un esempio potrebbe essere il repository “prova” dell’esempio precedente: come facciamo ad utilizzarlo se cambiamo pc sul quale lavoriamo?

Possiamo clonare un repository remoto con il comando

```
git clone urlRepositoryRemoto
```

E verrà creato, all’interno della directory corrente, il progetto remoto

Nel caso dell’esempio precedente:

```
git clone https://github.com/allemonta/prova.git
```

# GitHub - esempio [ clonazione ]

3 commits   1 branch   0 packages   0 releases   0 contributors

Branch: master   New pull request   Create new file   Upload files   Find file   Clone or download

Alessandro Aggiunto feature menu	
README.md	initial commit
about.html	Aggiunto pagina di about e collegata all'index
index.html	Aggiunto feature menu

Clone with HTTPS   Use SSH

Use Git or checkout with SVN using the web URL.

<https://github.com/allemonta/prova.git>

Open in Desktop   Open in Visual Studio   Download ZIP

```
MINGW64:/c:/Users/allem/OneDrive/Desktop
allem@DESKTOP-VBMRNV8 MINGW64 ~/OneDrive/Desktop
$ git clone https://github.com/allemonta/prova.git
Cloning into 'prova'...
remote: Enumerating objects: 14, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 14 (delta 5), reused 10 (delta 4), pack-reused 0
Unpacking objects: 100% (14/14), done.
```



# GitHub - aggiornamento

La piattaforma di GitHub contiene le modifiche che sono state inviate dai vari dispositivi. Per aggiornare il proprio repository locale al commit più recente presente nel repository remoto si utilizza il comando:

```
git pull
```

Se si vuole invece eliminare tutti i cambiamenti e commit eseguiti in locale, per poi sincronizzarsi con l'ultima versione presente sul repository remoto:

```
git fetch origin
```

```
git reset --hard origin/master
```

# GitHub - esempio [ altro progetto ]

Oltre ai passaggi riportati nella slide 54 c'è la possibilità muoversi nella seguente maniera:

1. Creare un nuovo repository su GitHub
2. Clonare il repository in locale → `git clone urlRepositoryRemoto`
3. Modificare il progetto in locale → `git add .` → `git commit -m "commento"`
4. Inviare le modifiche in remoto → `git push`



.gitignore



README.md

Montanari Alessandro  
A.S. 2020/2021

# GitHub - README.md

Un file README è un file di testo che introduce e spiega un progetto. Contiene informazioni comunemente richieste per comprendere di cosa tratta il progetto. Nella pagina del progetto di GitHub viene renderizzato, se esiste, il file README.md (che deve essere contenuto nella cartella principale del progetto).


Il testo contenuto dentro il file deve avere una formattazione particolare. Per esempio, per scrivere un titolo si deve scrivere:

```
# Progetto straordinario
```

Pagina di aiuto per la stesura del README: <https://www.makeareadme.com/>

# GitHub - README.md

README.md



## Material-UI pickers

Accessible, customizable, delightful date & time pickers for @material-ui/core

npm v3.2.10

downloads 1.1M/month

codecov 52%

minzipped size 18.2 KB

PASSED

cypress.io tests

visual testing

percy

code style prettier

### Installation

Note that this package requires @material-ui/core v4. It will not work with the old v3. Please read the [migration guide](#) if you are updating from v2

```
// via npm
npm i @material-ui/pickers

// via yarn
yarn add @material-ui/pickers
```

### Getting started

[Here is instruction](#) of how to get started with @material-ui/pickers .

# Git - .gitignore

Il file gitignore specifica intenzionalmente i file che non devono essere tracciati (e quindi da lasciare untracked) da Git. Ogni riga specifica un file specifico o un gruppo di file attraverso uno specifico pattern.

Qualche esempio:

- Per non tracciare il file “passwordSegrete.txt” presente nella cartella src  
`src/passwordSegrete.txt`
- Per non tracciare tutti i file con estensione “.old”  
`*.old`
- Per non tracciare tutti i file contenuti nella cartella src  
`src/`

Link utile per i pattern di .gitignore: <https://www.atlassian.com/git/tutorials/saving-changes/gitignore>



Montanari Alessandro  
A.S. 2020/2021

# GitHub pages

GitHub offre la possibilità di hostare gratuitamente, direttamente su un proprio repository un sito web.

Non si deve quindi creare altri account, comprare un dominio, comprare uno spazio web, impostare le credenziali per l'FTP o l'SFTP...

[Per creare la propria pagina su GitHub:](#)

1. Creare la repository "username.github.io", dove username **deve** essere l'username del proprio account GitHub
2. Clonare il repository in locale

```
git clone https://github.com/username/username.github.io
```



# GitHub pages

3. Aggiungere alla directory “username.github.io” appena creata i file del sito web. Si ricorda che quando si tenta di accedere alla pagina principale di un sito viene restituito il contenuto del file “index.html”
4. Aggiungere i file → `git add .`
5. Fare il commit → `git commit -m “initial commit”`
6. Inviare al server le modifiche → `git push`

Se tutto viene eseguito correttamente, si potrà raggiungere, dopo qualche secondo, la pagina contenente il sito web:

`https://username.github.io/`

Ora inviando altri file o modificando quelli esistenti nel repository remoto si modificherà il sito!