

Git & GitFlow

Come versionare codice in modo efficiente

Introduzione

Cos'è Git?

È un software

È un Version Control System (VSC)

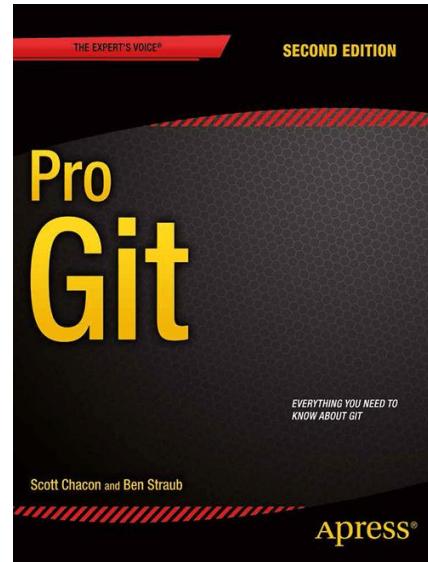
È un VCS distribuito

È un must per i programmatori moderni

È semplice quando vengono capite le basi

Cos'è un Version Control System?

"Un VCS è un sistema che registra, nel tempo, i cambiamenti ad un file o ad una serie di file, così da poter richiamare una specifica versione in un secondo momento."



Scott Chacon and Ben Straub, Pro Git

VCS Locale

Copiare incollare le versioni del progetto da una cartella in un'altra

Vantaggi:

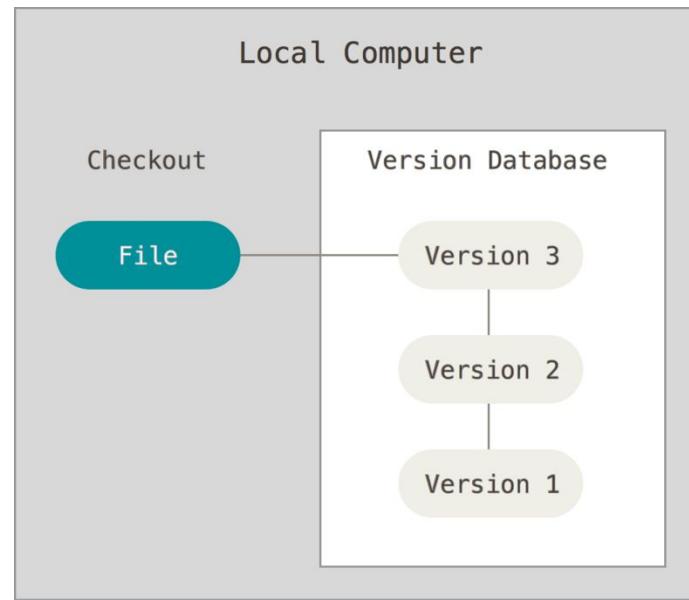
- * Semplicità

Svantaggi:

- * Comunemente soggetto a errori
- * Difficile da mantenere nel lungo tempo

Usi:

- * RCS -> usato in molti sistemi di backup di file



VCS Centralizzato

Unico server che contiene tutte le versioni e un numero di utenti che scaricano i file dal server centrale

Vantaggi:

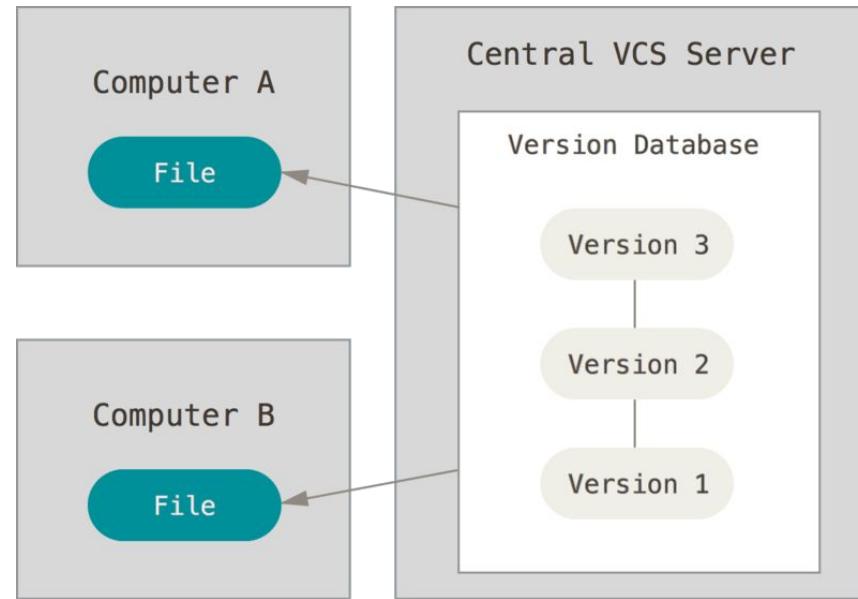
- * Sempre aggiornato
- * Amministratori hanno più controllo

Svantaggi:

- * Server

Usi:

- * Apache Subversion
- * Concurrent Versions System



VCS Distribuito

Ogni client contiene tutta la storia del progetto contenuto nel server.

Vantaggi:

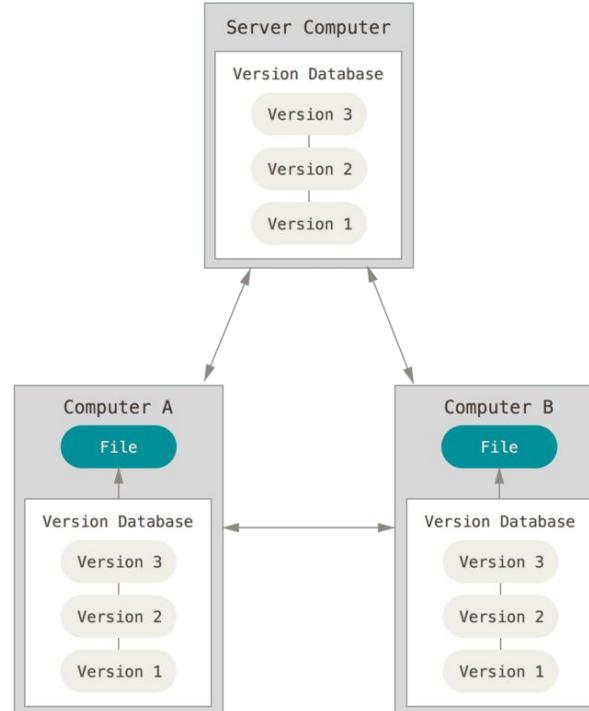
- * Serverless
- * Sicurezza

Svantaggi:

- * Curva di apprendimento lenta

Usi:

- * Git
- * Mercurial



Perchè usare Git

- È uno strumento completo
- Sta diventando uno standard
- Appartiene alla famiglia FOSS (Free and Open Source Software)

The screenshot shows the GitHub repository page for 'git / git'. At the top, there are buttons for 'Watch' (2.3k), 'Star' (34.7k), and 'Fork' (20.2k). Below this, tabs include 'Code' (selected), 'Pull requests' (40), 'Actions', 'Security' (11), and 'Insights'. The main area displays a list of commits from the 'master' branch. Key commits shown include:

Author	Commit Message	Date	Commits
gitster	First batch	14 hours ago	60,846
.github	ci: github action - add check for whitespace errors	19 days ago	
Documentation	First batch	14 hours ago	
block-sha1	sha1: provide another level of indirection for the SHA-1 functions	5 years ago	
builtin	Merge branch 'dl/checkout-guess'	14 hours ago	
ci	ci: do not skip tagged revisions in GitHub workflows	20 days ago	
compat	compat/mingw.h: drop extern from function declaration	21 days ago	

On the right side, there's an 'About' section with the following text:

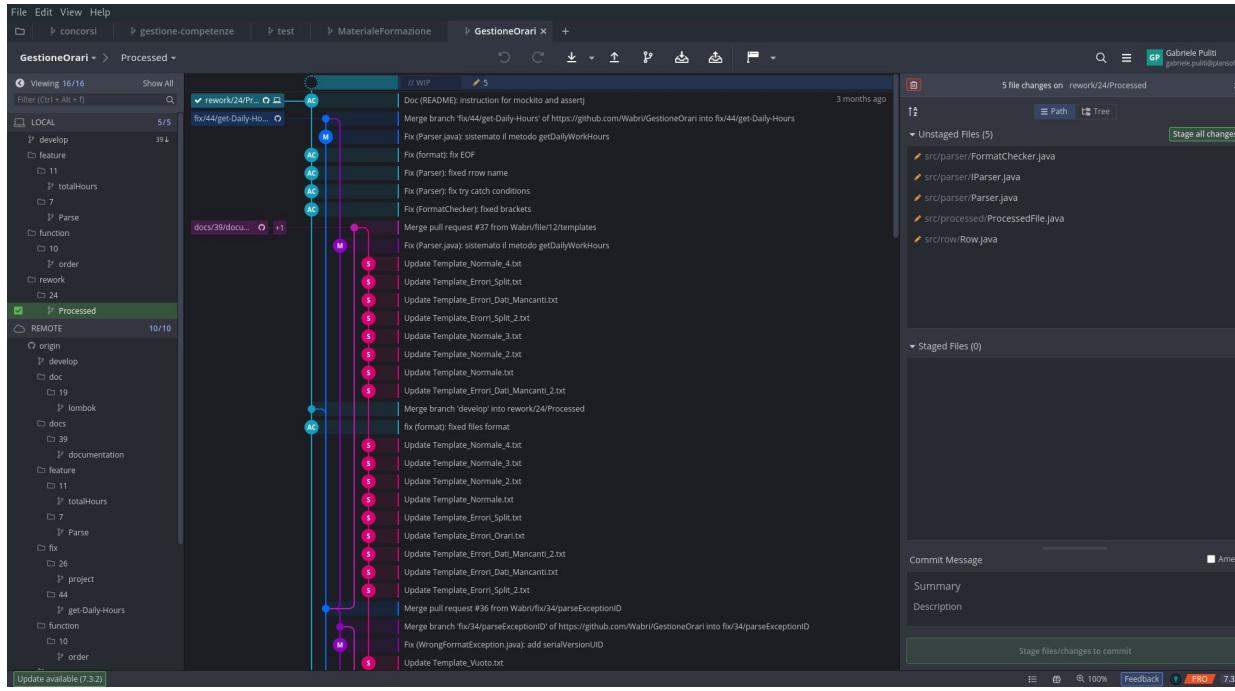
Git Source Code Mirror - This is a publish-only repository and all pull requests are ignored. Please follow Documentation/SubmittingPatches procedure for any of your improvements.

Below the 'About' section are links for 'c', 'shell', and 'hacktoberfest'. There are also links for 'Readme' and 'View license'.

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

Come usare Git

- GUI (Graphic User Interface)



Come usare Git

- GUI (Graphic User Interface)
- Terminale ⇐ Fortemente consigliato inizialmente

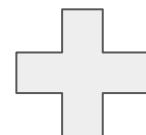
```
| Author: Antonio Caia <antonio.caia96@gmail.com>
| Date: Thu Jun 25 12:55:01 2020 +0200
|
|   Fix (Parser): fix try catch conditions
|
|   add WrongFormatException in try catch
|
* commit c2250ce26ce55358dd1d02bae32e1cfddd740400d
| Author: Antonio Caia <antonio.caia96@gmail.com>
| Date: Thu Jun 25 12:45:36 2020 +0200
|
|   Fix (FormatChecker): fixed brackets
|
* commit 547158c987d271004dfcf84e31e62aa8a0ea5b32
| Merge: 49cf5a4 af3dbaf
| Author: antoniocaia <>8995871+antonioacia@users.noreply.github.com>
| Date: Mon Jun 22 15:48:37 2020 +0200
|
|   Merge branch 'develop' into rework/24/Processed
|
* commit af3dbafc27242f56579be992feef372b17b1daa3
| Merge: 8c4aadd 82368d
| Author: MariusLovesPizza <15158862+MariusLovesPizza@users.noreply.github.com>
| Date: Mon Jun 22 14:58:49 2020 +0200
|
|   Merge pull request #36 from Wabri/fix/34/parseExceptionID
|
|   34 - Fix (WrongFormatException.java): add serialVersionUID
|
75% 12:54:54 ~/Workspace/academy/GestioneOrari ↵ rework/24/Processed +5 ✘
⌚ > git status
On branch rework/24/Processed
Your branch is up to date with 'origin/rework/24/Processed'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:  src/parser/FormatChecker.java
    modified:  src/parser/IParser.java
    modified:  src/parser/Parser.java
    modified:  src/processed/ProcessedFile.java
    modified:  src/row/Row.java

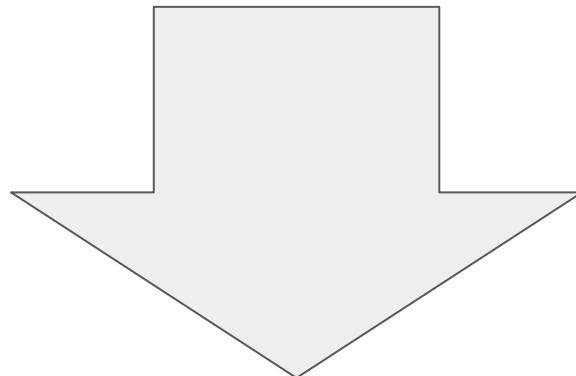
no changes added to commit (use "git add" and/or "git commit -a")
75% 12:54:57 ~/Workspace/academy/GestioneOrari ↵ rework/24/Processed +5 ✘
⌚ > git add src/
75% 12:55:02 ~/Workspace/academy/GestioneOrari ↵ rework/24/Processed +5 ✘
⌚ > █
```

Qual è la soluzione migliore?

GUI



Terminale



Ibrido

Ciclo Vita Versionamento

Stati di un file in un VCS distribuito

- Untracked
- Dirty
- Staged
- Clean

Ogni VCS potrebbe utilizzare una diversa nomenclatura per gli stati.

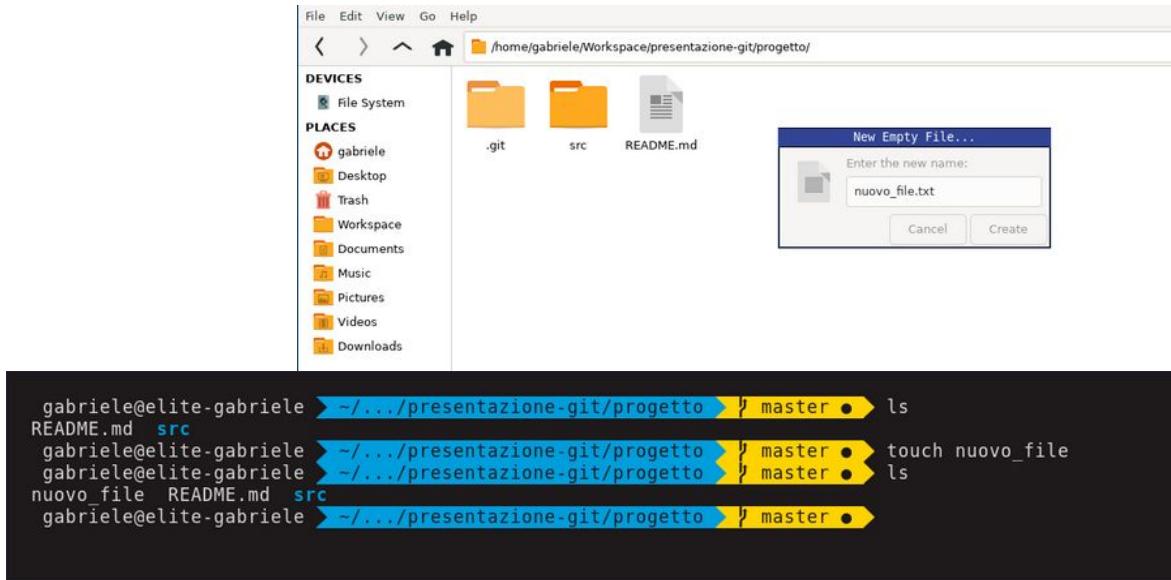
Per questo motivo in git abbiamo:

- Untracked
- Modified = Dirty
- Staged
- Unmodified = Clean

Untracked => Nuovo file **non** versionato

Quando

È necessario **aggiungere** un file **non** preesistente nel progetto



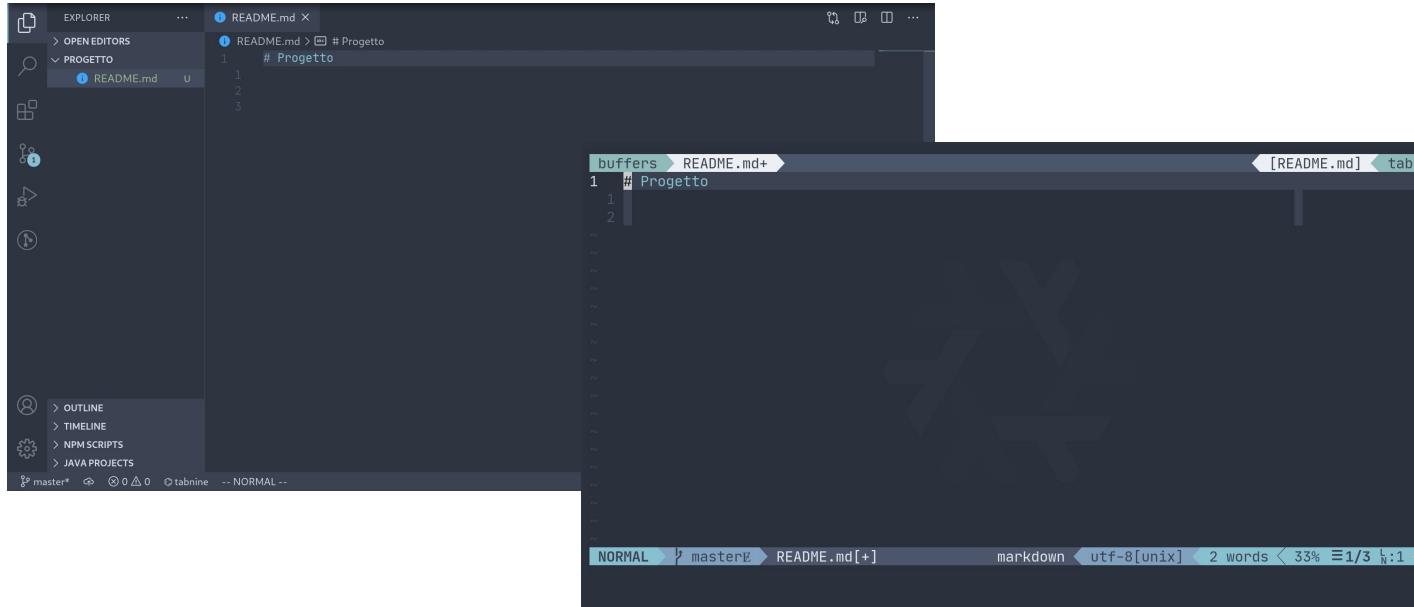
The screenshot shows a desktop environment with a file manager window open. The file manager displays a directory structure with a '.git' folder, a 'src' folder, and a 'README.md' file. A 'New Empty File...' dialog box is overlaid on the file manager, prompting the user to enter a new file name. The text 'nuovo_file.txt' is entered in the dialog. Below the file manager, a terminal window shows the following command history:

```
gabriele@elite-gabriele:~/.../presentazione-git/progetto$ ls
README.md  src
gabriele@elite-gabriele:~/.../presentazione-git/progetto$ touch nuovo_file
gabriele@elite-gabriele:~/.../presentazione-git/progetto$ ls
nuovo_file README.md  src
gabriele@elite-gabriele:~/.../presentazione-git/progetto$
```

Modified => Modifica di un file **versionato**

Quando

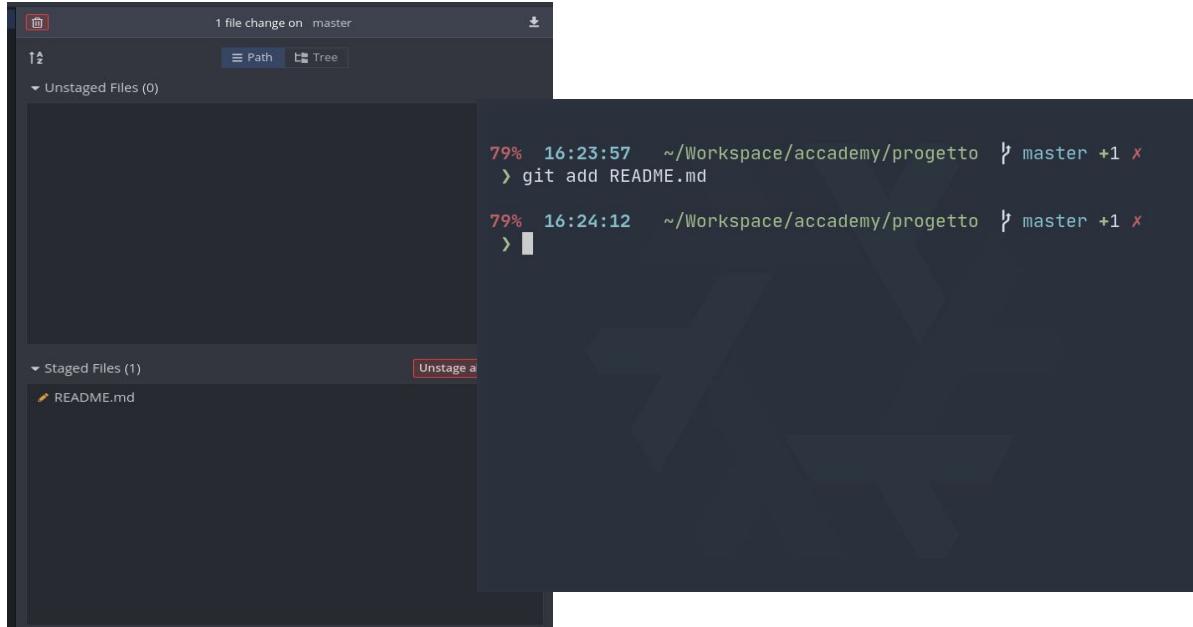
È necessario **modificare** un file **già** preesistente nel progetto



Staged => File pronto per essere **versionato**

Quando

Si vuole salvare lo stato del file generando una nuova versione.



The screenshot shows a terminal window with the following output:

```
79% 16:23:57 ~/Workspace/accademy/progetto ↵ master +1 ✘
> git add README.md

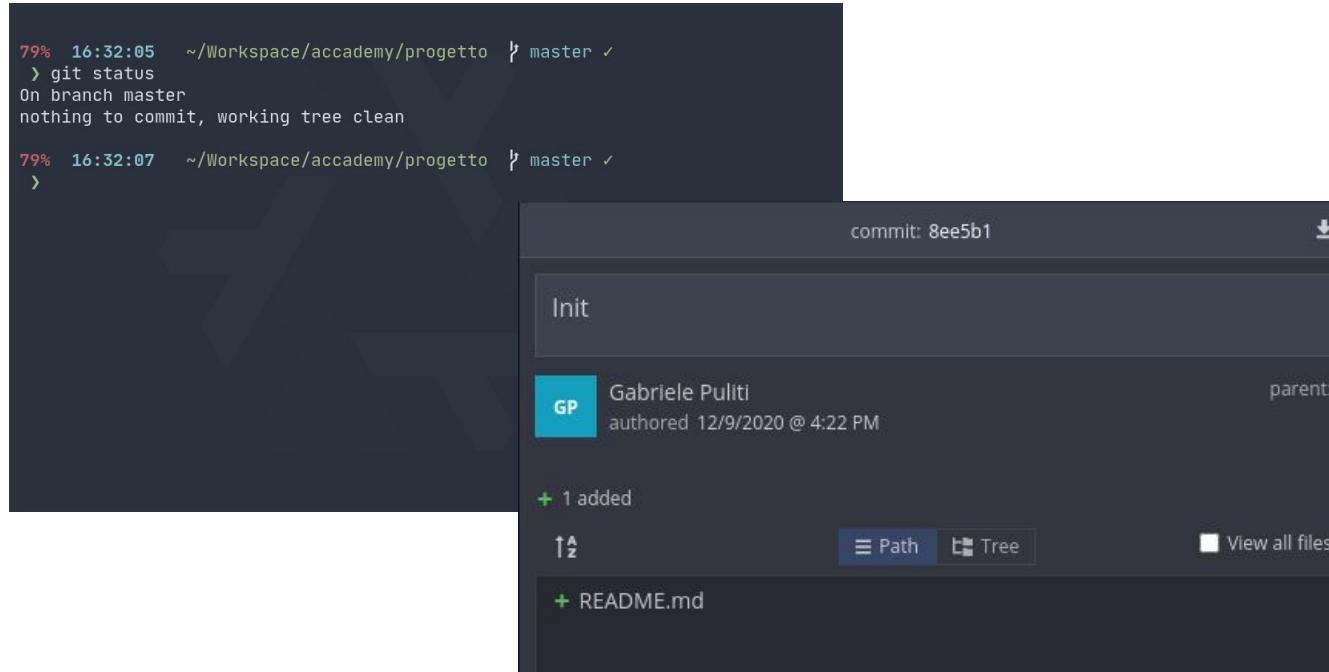
79% 16:24:12 ~/Workspace/accademy/progetto ↵ master +1 ✘
>
```

The terminal is running on a Linux system, indicated by the prompt (~). The command `git add README.md` was run to stage the `README.md` file. The output shows the progress (79%) and the current status of the repository (master branch, 1 change staged).

Unmodified => File **versionato**

Quando

Il file è in linea con l'ultima versione del progetto



The image shows a terminal window and a GitHub commit history side-by-side.

Terminal:

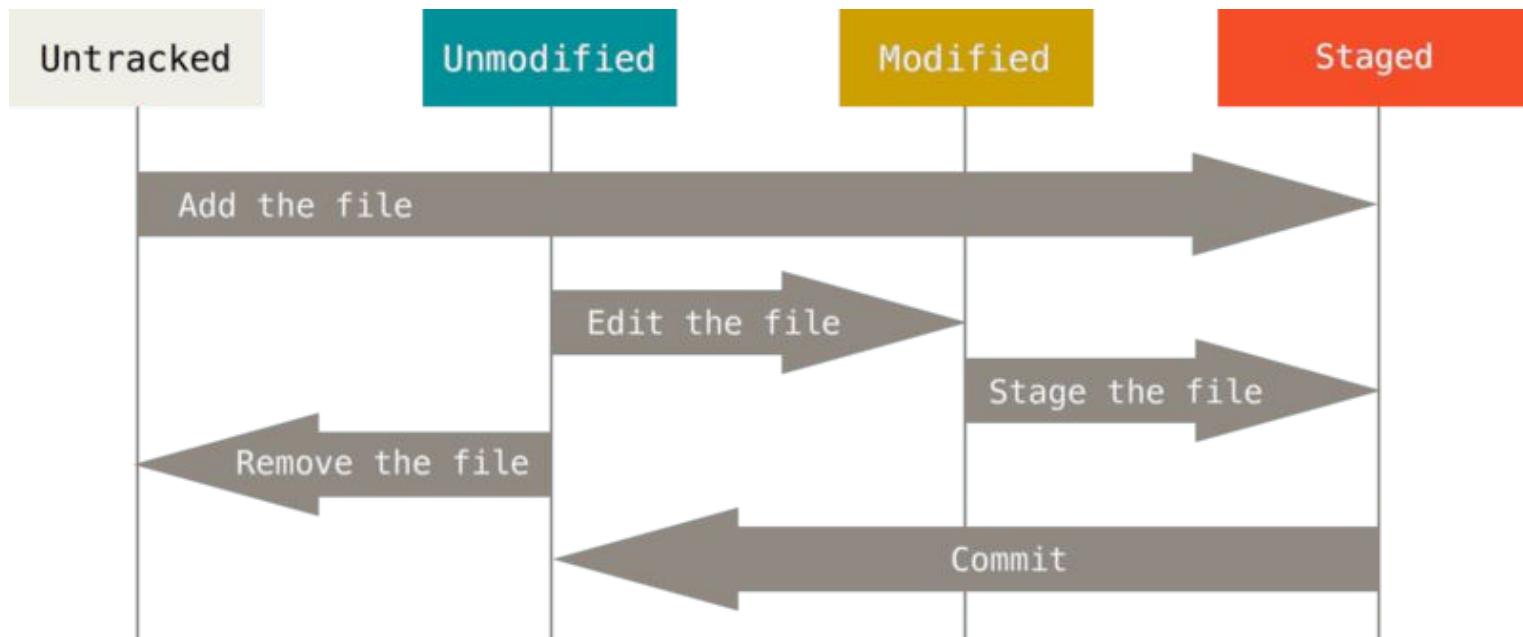
```
79% 16:32:05 ~/Workspace/accademy/progetto ✘ master ✓
> git status
On branch master
nothing to commit, working tree clean

79% 16:32:07 ~/Workspace/accademy/progetto ✘ master ✓
>
```

Github Commit History:

- commit: 8ee5b1
- Init
- GP Gabriele Puliti authored 12/9/2020 @ 4:22 PM parent:
- + 1 added
- README.md

Schema transizioni dello stato dei file



Come si usa

Come si usa git

Git può essere usato in molti modi: terminale, interfaccia grafica e direttamente da un editor o un ide.

Git è utilizzato anche per questa sua natura di essere versatile e portatile in ogni ambiente di sviluppo.

Usare git significa eseguire delle funzionalità specifiche che chiameremo comandi.

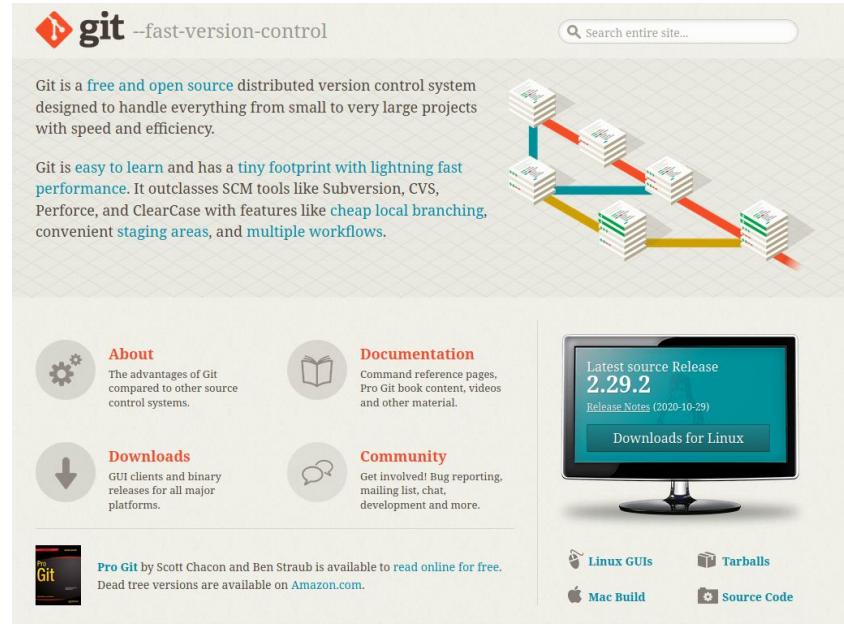
Esistono all'incirca 100 comandi nel compendio di git e ogni comando ha dei sottocomandi quindi si potrebbe arrivare con facilità alle 1000 funzionalità se vogliamo contarle singolarmente.

È necessario conoscerle tutte? No.

Come installare git - terminale

Per usare git da terminale, scelta consigliata per apprendere a pieno lo strumento, basta installare il software direttamente dal sito:

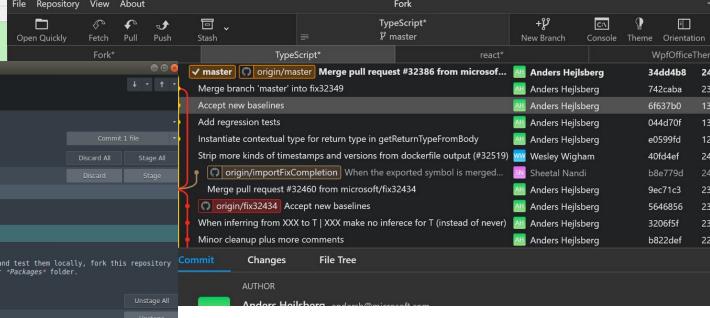
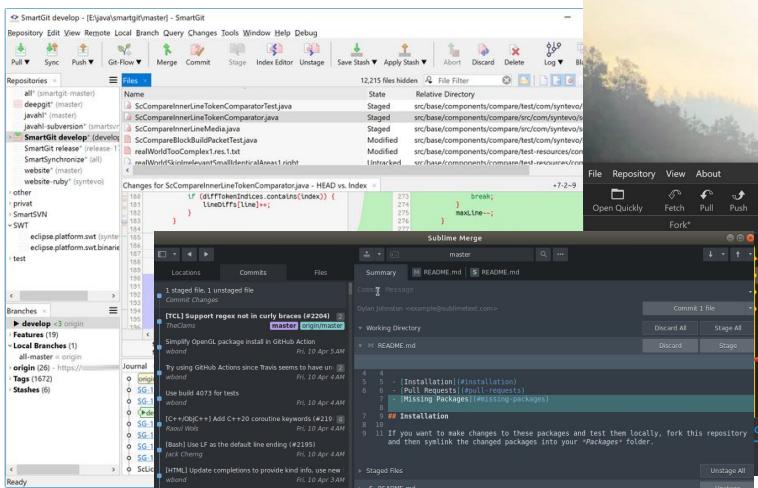
<https://git-scm.com>



Come installare git - GUI

Di strumenti grafici è possibile trovarne tantissimi, ne cito qualcuno: GitKraken, Sourcetree, Smartgit, Sublime Merge...

Potete trovare una lista completa nel sito di git:
<https://git-scm.com/downloads/guis>



A composite screenshot showing two Git GUI tools. On the left is Sourcetree, displaying a commit history graph with several branches and commits. On the right is the GitKraken interface, which includes a commit history view and a large promotional banner for the tool. The banner features the GitKraken logo and the text "Legendary Git GUI client for Windows, Mac & Linux". Below the banner are sections for "Download for Free", "Other Platforms", and "See Features".

Consiglio

Inizialmente usare la linea di comando.

Git è software pensato per essere usato a linea di comando.

Le interfacce grafiche imitano il workflow che viene usato a linea di comando.

Comprendere git a linea di comando facilità l'uso delle interfacce grafiche e dei comandi più complessi.

Alcuni concetti

Definizioni

Repository = il progetto su cui stiamo lavorando

Storia della repository = Tutte le modifiche salvate del progetto

Remoto = Progetto che si trova su un server non locale

Branch = Linee temporali del progetto

I primi comandi

I primi 4 comandi

- Init
- Add
- Commit
- Status

Init => Inizializzazione del progetto

Quando	Si vuole creare un nuovo progetto
Come	Si crea una cartella, ci spostiamo dentro la cartella e si esegue il comando init
Cosa fa	Genera tutti i file utilizzati da git: file di configurazione, logs, refs, hooks, etc.etc.

Init => Sintassi

```
git init
```

Questo comando deve essere eseguito all'interno della cartella rappresentante il progetto che vogliamo versionare.

Ricordarsi quindi di creare la cartella di destinazione, se non esiste già, e spostarsi all'interno di essa.

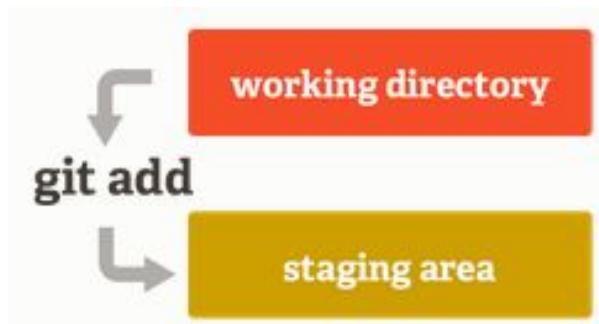
Add => Modificare lo stato di un file

Quando

Si vuole inserire nella storia del progetto un file (o più)

Cosa fa

Sposta lo stato di un file da **Modified/Untracked** allo stato **Staged**



Add => Sintassi

```
git add <percorso_file>  
git add <percorso_cartella>
```

Il sottocomando add può assumere un file o un'intera cartella. Nel primo caso sposta solo quel file, nel secondo tutti i file che si trovano in quella cartella.

Alcuni esempi:

```
git add src/main/application.java  
git add src/main/
```

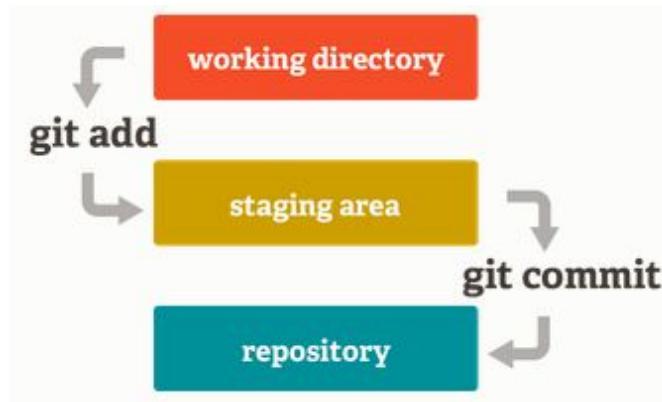
Commit => Generare una nuova versione

Quando

Salvare nella storia del progetto le modifiche dei file

Cosa fa

Sposta lo stato di un file da **Staged** allo stato **Unmodified**, generando una nuova versione del progetto



Commit => Sintassi

```
git commit
```

Questo comando aprirà il vostro editor di sistema preferito, questo perchè per ogni nuova versione che aggiungiamo alla storia del progetto dobbiamo dare una descrizione più o meno specifica delle modifiche fatte.

Descrizione sbagliata

```
1 Modifiche
1
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 #
5 # On branch master
6 #
7 # Initial commit
8 #
9 # Changes to be committed:
10 #   new file:  nuovo_file.txt
11 #
```

Descrizione corretta

```
1 Aggiornato il progetto con un nuovo file
1
2 Il file contiene delle modifiche importanti per poter risolvere il
3 problema generato al servizio di creazione nuovi componenti.
4
5 Risolve #42
6
7 # Please enter the commit message for your changes. Lines starting
8 # with '#' will be ignored, and an empty message aborts the commit.
9 #
10 # On branch master
11 #
12 # Initial commit
13 #
14 # Changes to be committed:
15 #   new file:  nuovo_file.txt
16 #
```

Commit => Sintassi

```
git commit -m "Descrizione"
```

Questo comando **non** aprirà il vostro editor di sistema preferito, ma inserirà come descrizione del commit quello che viene inserito all'interno dei doppi apici.

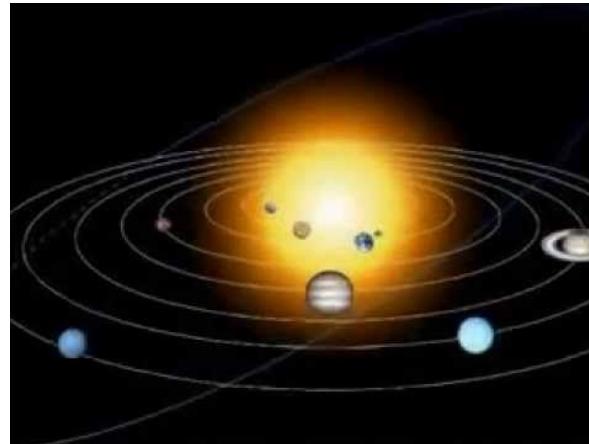
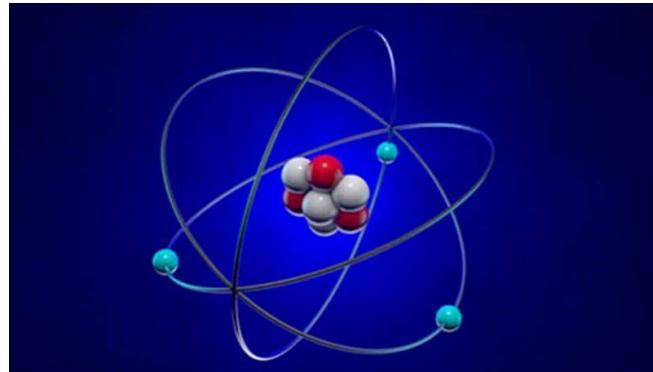
Questo modo di fare i commit per quanto sia comune, è sconsigliato. Più le descrizioni sono precise e più sarà facile ritrovare una modifica fatta.

Commit => approfondimento

Ai commit che vengono eseguiti con successo viene assegnato un identificativo alfanumerico univoco di 40 caratteri (generati usando un algoritmo chiamato **SHA-1**) che noi chiameremo id del commit o sha del commit.

Questo consente di avere un riferimento unico per ogni commit fatto.

Matematicamente parlando un repository può avere un massimo di commit pari a $36^{40} \approx 10^{57}$ = **numero di atomi nel sistema solare.**



Commit => amend

```
git commit --amend
```

```
git commit --amend -m "Descrizione"
```

Quando

Si sbaglia la descrizione **dell'ultimo commit** oppure se si è dimenticato di aggiungere un file

Cosa fa

Modifica **l'ultimo** versionamento del codice rigenerando un nuovo commit che sostituisce il precedente

Info

Per poter includere nel commit un nuovo file o nuova modifica a un file è necessario che prima di eseguire l'amend il file incriminato si trovi nello stato staged

Status => Visionare lo stato del progetto

Quando

Vogliamo verificare lo stato dei files del progetto

Cosa fa

Restituisce un riassunto dello stato del progetto con informazioni riguardanti lo stato dei file. I file che si trovano nello stato unmodified non vengono considerati.

Status => Sintassi

```
git status
```

Può restituire la lista dei file che si trovano nella stato:

- stage
- modified
- untracked

Oppure un output vuoto in caso di progetto aggiornato.

Status => Esempio

```
gabriele@elite-gabriele ~/.../presentazione-git/progetto master • git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   src/file.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    untracked

gabriele@elite-gabriele ~/.../presentazione-git/progetto master •
```

Esempio 1

Passo 1 => Creazione repository

1. Creare la cartella che conterrà il nostro progetto
2. Spostarsi nella directory
3. Inizializzare il progetto git

Completati questi steps il nostro progetto conterrà una directory chiamata .git contenente i file di configurazione e la storia del nostro progetto.

```
79% 09:29:11 ~/Documents
> mkdir progetto ; cd progetto ; git init ; ls -al
Initialized empty Git repository in /home/gab/Documents/progetto/.git/
drwxr-xr-x - gab 4 Dec 9:29 .git
```

Passo 2 => Primo commit

1. Aggiungere un file chiamato **README.md**
2. Versionare il file **README.md**

Questo rappresenterà il nostro primo commit e sarà la radice di tutto il progetto da cui discenderanno tutti i commits successivi.

```
79% 09:29:15 ~/Documents/progetto ↵ master ✓
> touch README.md ; git add README.md ; git commit
[master (root-commit) 444f4eb] Init project
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md
```

Passo 3 => Commit sbagliato

1. Aggiungere un **nuovo file** al progetto
2. Modificare il **README.md** inserendoci una frase
3. Sia il nuovo file che le modifiche apportate al README.md devono essere **incluse nel commit precedente**

Eseguiti questi 3 passi avremmo rigenerato un nuovo commit che andrà a sostituire il precedente.

```
79% 09:35:29 ~/Documents/progetto ✘ master ✓
> touch file.txt ; echo "# Progetto" > README.md ; git add README.md file.txt ; git commit --amend
[master 4b72558] Init project
Date: Fri Dec 4 09:35:05 2020 +0100
2 files changed, 1 insertion(+)
create mode 100644 README.md
create mode 100644 file.txt
```

Bonus 1 => Analisi output del commit

```
[master (root-commit) 444f4eb] Init project
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md
```

- Prima riga:
 - nome **branch**
 - **root** commit
 - **sha** commit
 - prima riga della **descrizione**
- Seconda riga:
 - **numero di file** inclusi nel commit
 - numero di **modifiche** apportate
- Terza riga:
 - **modalità** di aggiunta del file
 - codice associato alla **tipologia** del file
 - **nome** del file

Bonus 2 => Codice delle tipologie

codice 6 cifre	significato
040000	Directory
100644	File di testo
100664	File di testo non modificabile
100755	File eseguibile
120000	Symbolic Link
160000	Gitlink

Comunicare con il server

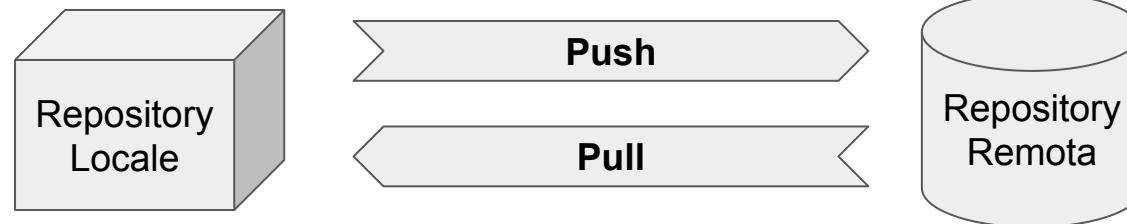
Comunicare con il server

- Remote
- Push
- Pull

Remote => Server remoti

Quando Gestire i collegamenti esterni

Cosa fa Genera dei collegamenti tra il repository



Remote => Sintassi per aggiungere un remoto

```
git remote add <nome_remote> <url_remote>
```

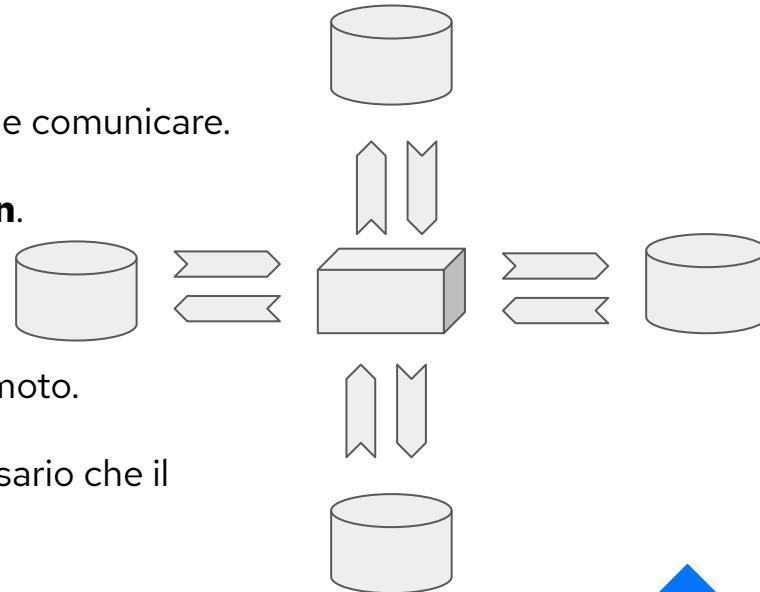
Questo comando aggiunge un **nuovo remoto** con cui è possibile comunicare.

Il nome del repository remoto solitamente viene chiamato **origin**.

È possibile avere **più remoti** collegati ad una repository.

Lo **URL** di riferimento corrisponde all'indirizzo del repository remoto.

Prima di collegare un remoto al nostro repository locale è necessario che il remoto esista.

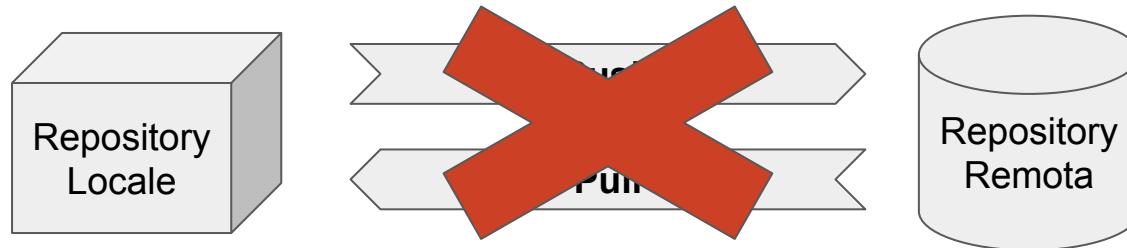


Remote => Sintassi per rimuovere un remoto

```
git remote remove <nome_remote>
```

Questo comando rimuoverà il collegamento con il repository esterno.

Non elimina però il repository remoto.



Come creare una repository remota

Una repository remota può essere creata su:

- Server privati
- Hosting online (github e gitlab)

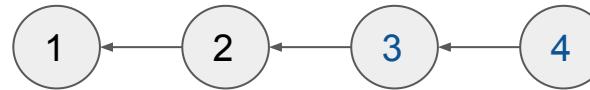
The screenshot shows a dark-themed GitLab interface. At the top, there is a navigation bar with a bell icon, a plus sign for creating new items, and a user profile picture. A dropdown menu is open over the plus sign, listing options: "New repository", "Import repository", "New gist", "New organization", and "New project". Below the menu, a terminal window displays the command "git init" being run in a directory named "ProgettoRemoto". The bottom part of the screen shows a "Projects" section with a "New project" button. A small blue diamond-shaped badge in the bottom right corner contains the number "51".

```
wabri@Fabulinus ~/.../Workspaces/ProgettoRemoto git init
Initialized empty Git repository in /home/wabri/Documents/Workspaces/ProgettoRemoto/.git/
wabri@Fabulinus ~/.../Workspaces/ProgettoRemoto master
```

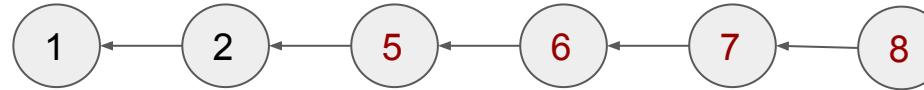
Branch => Linee temporali

- Ogni cerchio corrisponde a una nuova modifica
- Ogni nuova modifica è legata alla modifica fatta precedentemente
- Le modifiche (e il grafico sotto) si leggono da sinistra a destra

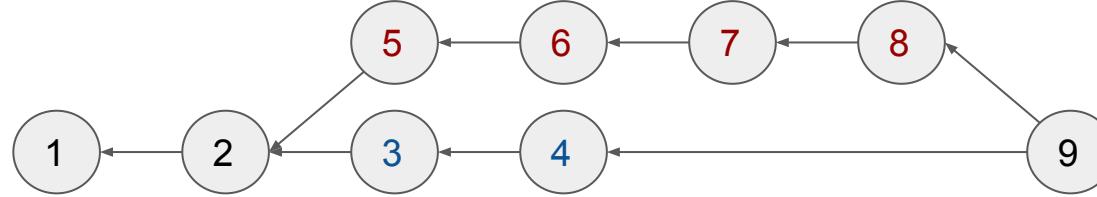
Linea temporale 1



Linea temporale 2



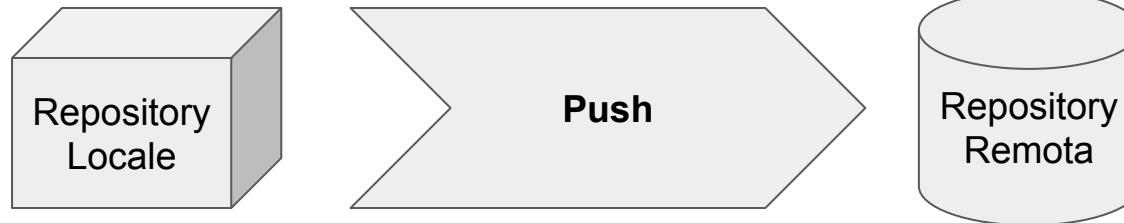
Linea temporale 3



Push => Aggiornare il remote

Quando Vogliamo aggiornare il repository **remoto** con le modifiche effettuate in **locale**

Cosa fa **Impacchetta le modifiche locali** e le invia al repository remoto aggiornandolo

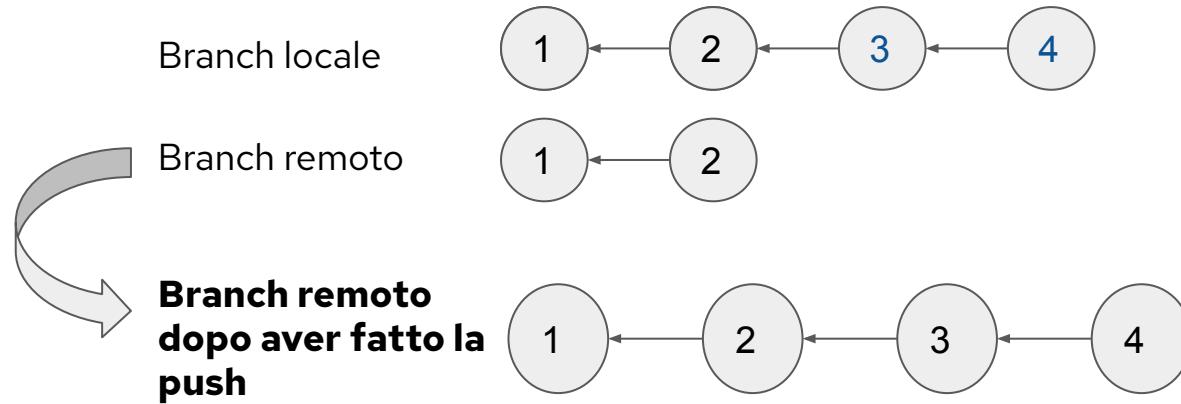


Push => Sintassi

```
git push <nome_remoto> <nome_branch>
```

Aggiorna il branch remoto con le modifiche del branch locale, se ce ne sono.

Consideriamo di aver fatto 2 commit in più sul branch locale:



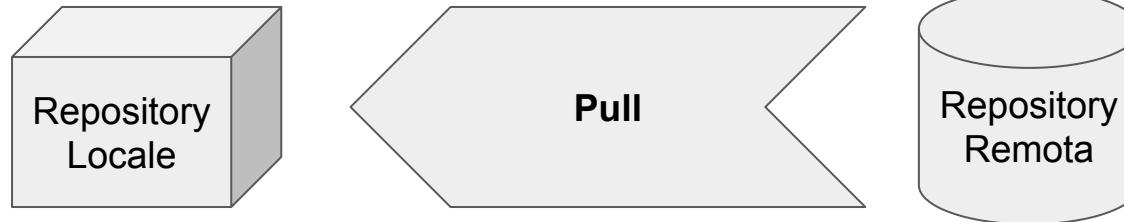
Pull => Aggiornare il locale

Quando

Vogliamo aggiornare il repository **locale** con le modifiche effettuate in **remoto**

Cosa fa

Scarica le modifiche remote e aggiorna il repository locale

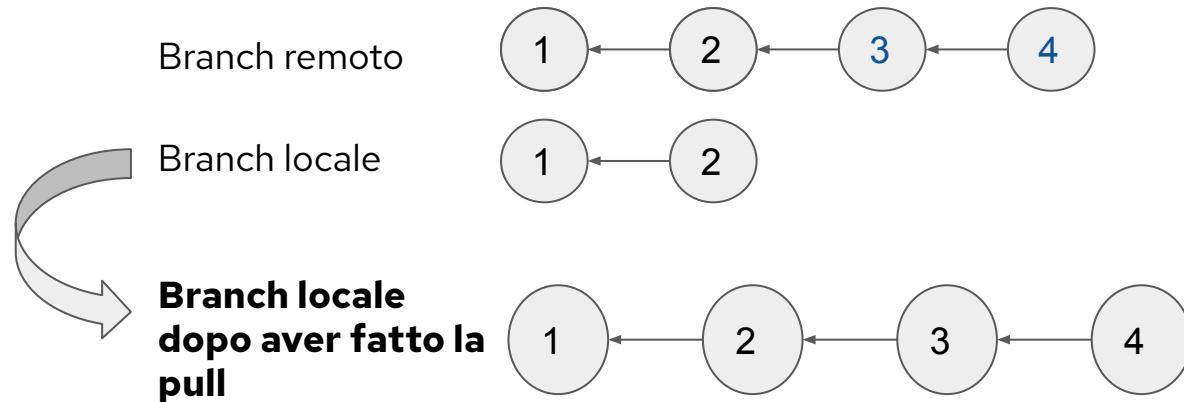


Pull => Sintassi

```
git pull <nome_remoto> <nome_branch>
```

Aggiorna il branch locale con le modifiche del branch remoto, se ce ne sono.

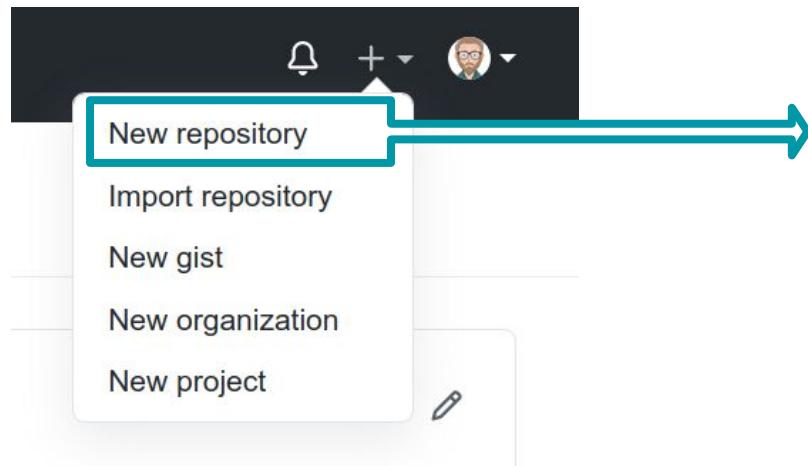
Consideriamo di aver fatto 2 commit in più sul branch remoto:



Esempio 2

Passo 1 => Creazione repository **remoto**

1. Avere un account su un **servizio di host**
2. Creare una **nuova 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 *

Wabri /

Great repository names are short and memorable. Need inspiration? How about [special-octo-eureka?](#)

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](#)

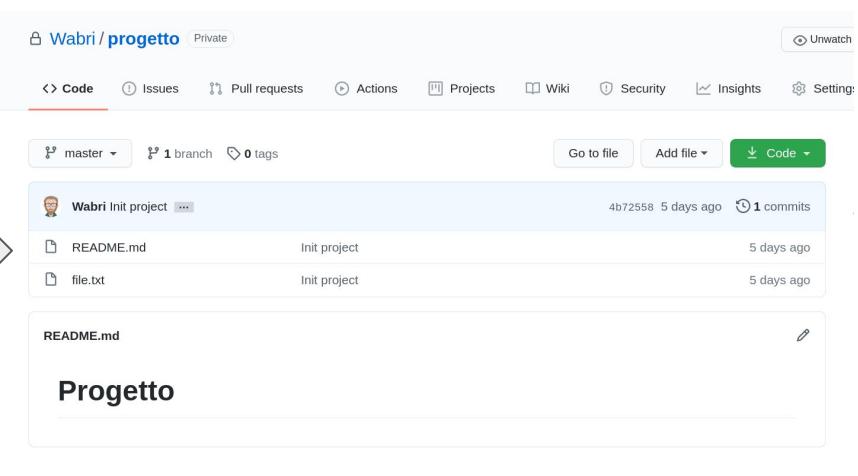
Passo 2 => Aggiungere il remoto

1. Prendere il progetto fatto nell'esempio 1
2. Aggiungere il remoto chiamandolo **origin**
3. Eseguire la **push** e osservare le modifiche in remoto

```
79% 16:56:24 ~/Documents/progetto ⌘ master ✓
> git remote add origin git@github.com:Wabri/progetto.git

79% 16:56:26 ~/Documents/progetto ⌘ master ✓
> git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 331 bytes | 331.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:Wabri/progetto.git
 * [new branch]      master -> master

79% 16:56:35 ~/Documents/progetto ⌘ master ✓
>
```



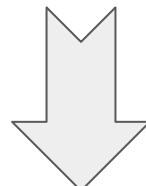
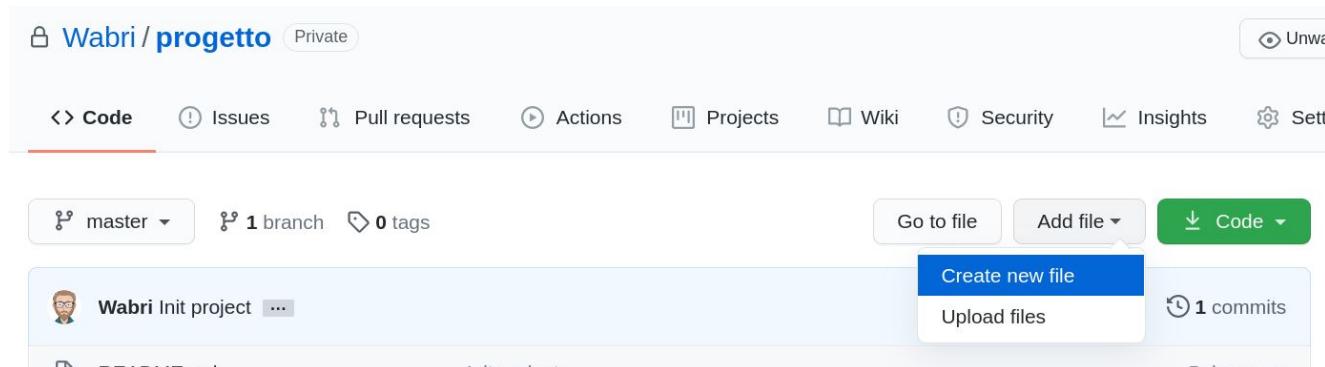
Bonus 1 => Analisi output della push

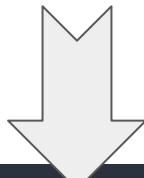
```
> git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 331 bytes | 331.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:Wabri/progetto.git
 * [new branch]      master → master
```

- Primo Blocco:
 - Conteggio elementi
 - Compressione elementi
- Secondo Blocco:
 - Upload archivio compresso
 - Stato delle modifiche effettuate nell'origin

Passo 3 => Modificare il remoto

1. Aggiungere un **nuovo file** nel remoto (usando il sito) chiamato **nuovo_file.txt** e inserire una qualche frase
2. Eseguire la **pull** per ottenere le modifiche in locale





```
79% 17:28:19 ~/Documents/progetto ↵ master ✓
⌚ > git pull origin master
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 709 bytes | 709.00 KiB/s, done.
From github.com:Wabri/progetto
 * branch           master    → FETCH_HEAD
   4b72558..ad26ebc  master    → origin/master
Updating 4b72558..ad26ebc
Fast-forward
  nuovo_file.txt | 1 +
  1 file changed, 1 insertion(+)
  create mode 100644 nuovo_file.txt
```

```
79% 17:28:32 ~/Documents/progetto ↵ master ✓
⌚ > cat nuovo_file.txt
```

	File: nuovo_file.txt
1	Questo è un nuovo file

```
79% 17:28:38 ~/Documents/progetto ↵ master ✓
⌚ > █
```

Bonus 2 => Analisi output della pull

```
git pull origin master
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 709 bytes | 709.00 KiB/s, done.
From github.com:Wabri/progetto
 * branch           master      → FETCH_HEAD
   4b72558..ad26ebc  master      → origin/master
Updating 4b72558..ad26ebc
Fast-forward
  nuovo_file.txt | 1 +
  1 file changed, 1 insertion(+)
  create mode 100644 nuovo_file.txt
```

- Primo blocco:
 - Recupero delle modifiche nel remoto
- Secondo blocco:
 - Aggiornamento della versione della storia locale
- Terzo blocco:
 - Sommario delle modifiche recuperate del remoto origin

Cominciare a collaborare

Cominciare a comunicare

- Clone
- Branch
- Checkout
- Merge

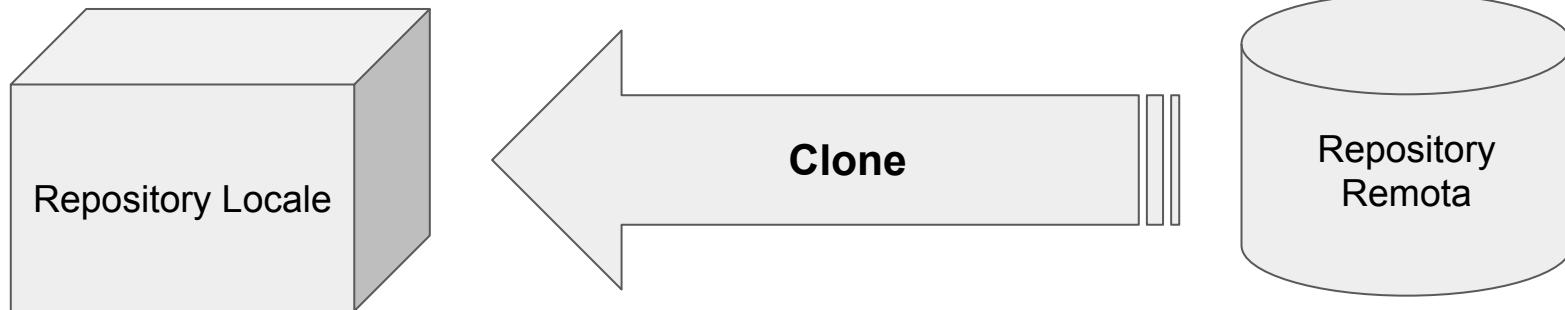
Clone => Copiare repository remoto

Quando

Vogliamo recuperare un progetto che si trova in un host remoto

Cosa fa

Scarica tutta la storia del repository remoto e inizializza il progetto locale



Clone => Sintassi

```
git clone <url>
```

Questo comando permette di scaricare la repository che si trova in remoto e generare un clone perfettamente uguale localmente.

Lo url corrisponde all'indirizzo in cui si trova la repository, quindi potrebbe essere un indirizzo github o una directory che si trova in un server a cui posso accedere tramite ssh.

È possibile anche rinominare la directory che conterrà il progetto:

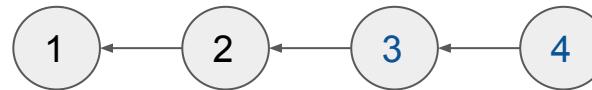
```
git clone <url> <nome>
```

Di default il nome della directory assumerà il nome della repository in cui si trova il progetto.

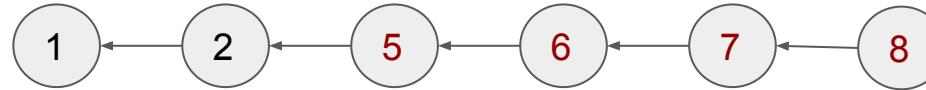
Branch => Linee temporali

- Ogni cerchio corrisponde a una nuova modifica
- Ogni nuova modifica è legata alla modifica fatta precedentemente
- Le modifiche (e il grafico sotto) si leggono da sinistra a destra

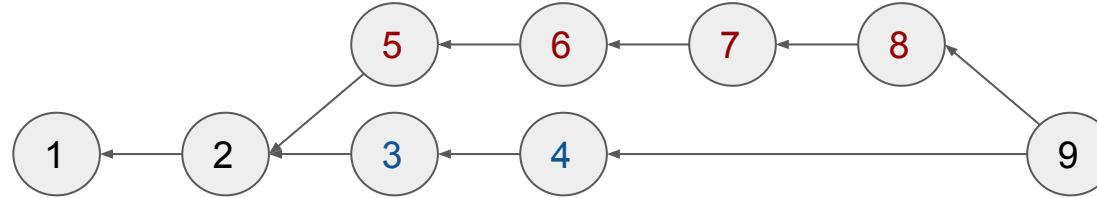
Linea temporale 1



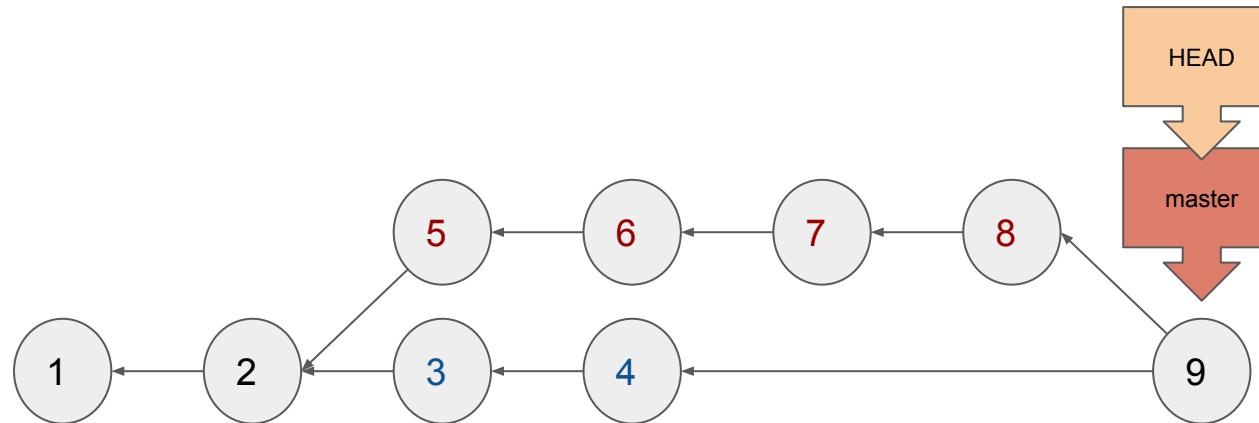
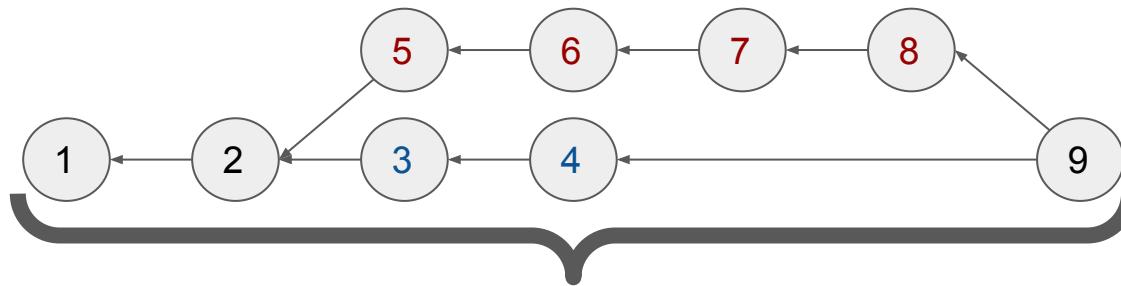
Linea temporale 2

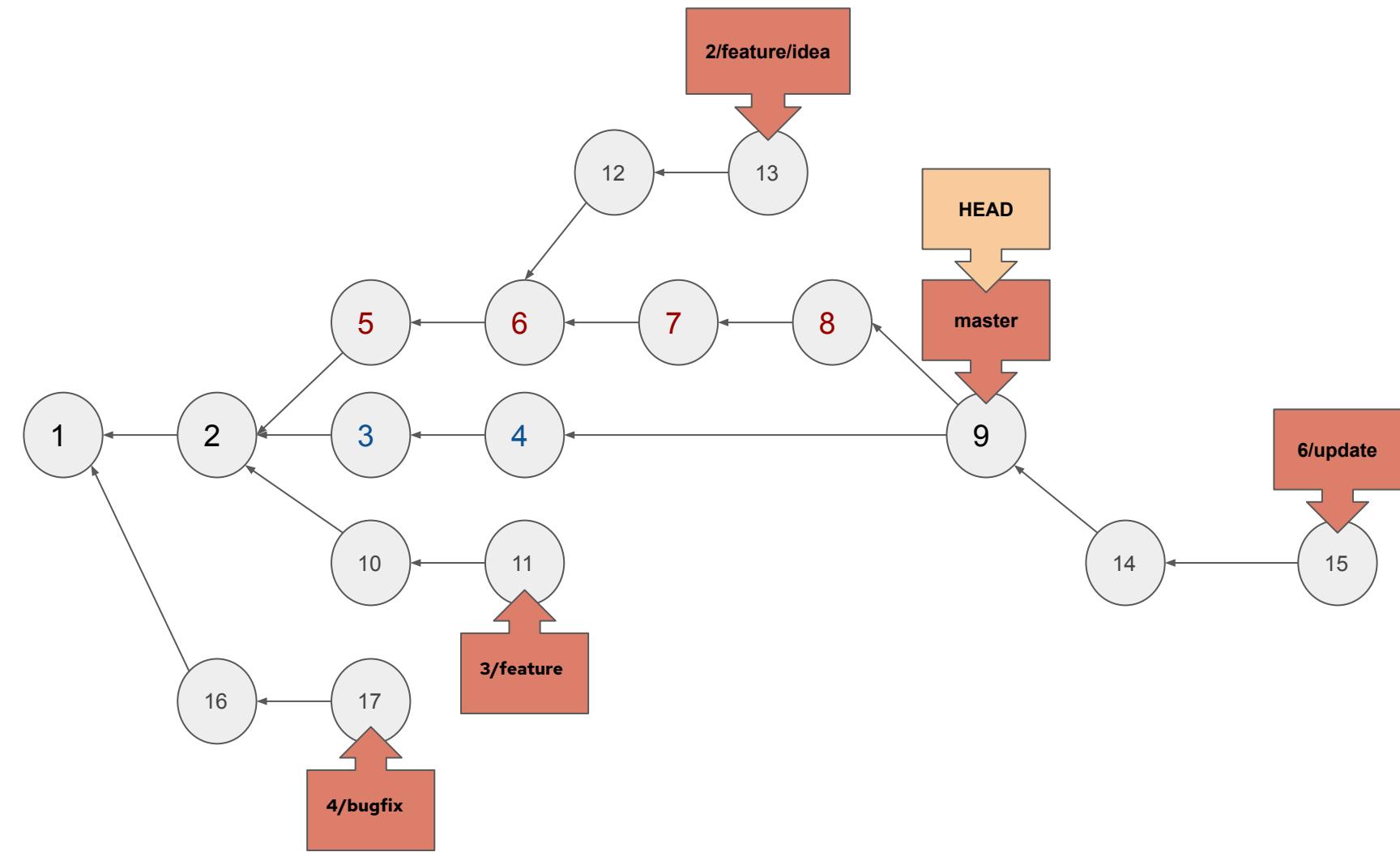


Linea temporale 3



Branch => Puntatori

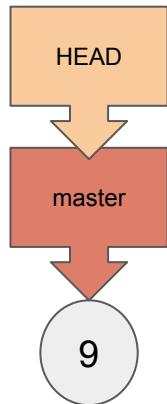




Branch => HEAD

Esiste un puntatore speciale chiamato **HEAD** che è il modo in cui riusciamo a spostarci nella storia.

Dependent/Attached HEAD



Detached HEAD



Possiamo spostarci liberamente nella storia della repository usufruendo sia dei branch sia degli sha dei commit.

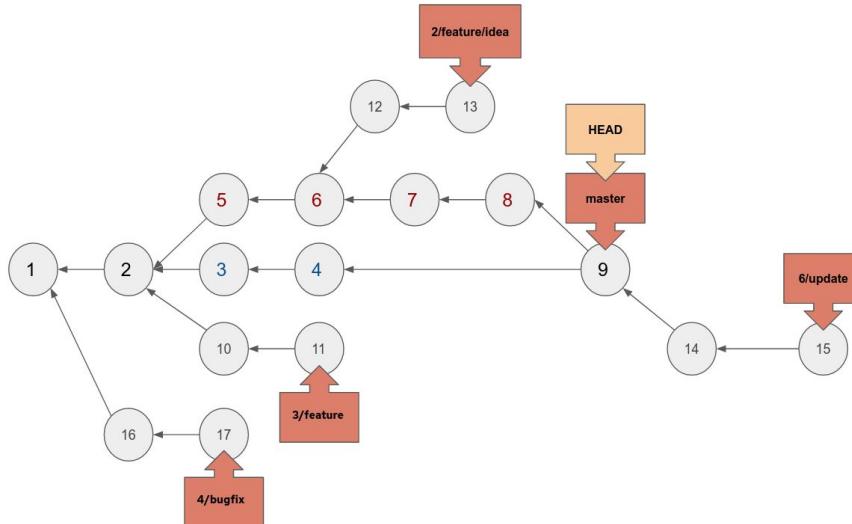
Branch => Il comando

Quando

Vogliamo aggiungere, recuperare o eliminare un branch.

Cosa fa

Gestisce i branch della repository



Branch => Sintassi

```
git branch <nome_branch>
```

Questo comando permette di creare un nuovo branch a partire dalla posizione in cui ci troviamo, quindi genera un branch a partire dal commit/branch a cui sta puntando **HEAD**.

Questo comando gestisce i branch nel repository locale, quindi non influisce in alcun modo sul repository remoto.

Tutte le modifiche che vengono fatte con questo comando possono essere trasferite su remoto usando il comando push precedentemente definito.

Branch => Usi

```
git branch
```

Questo comando restituirà la lista di tutti i branch attualmente presenti nel repository **locale**

```
git branch <nome_branch> <sha_commit>
```

Con questo comando verrà **creato** un nuovo branch a partire dal commit con il codice indicato

```
git branch -d <nome_branch>
```

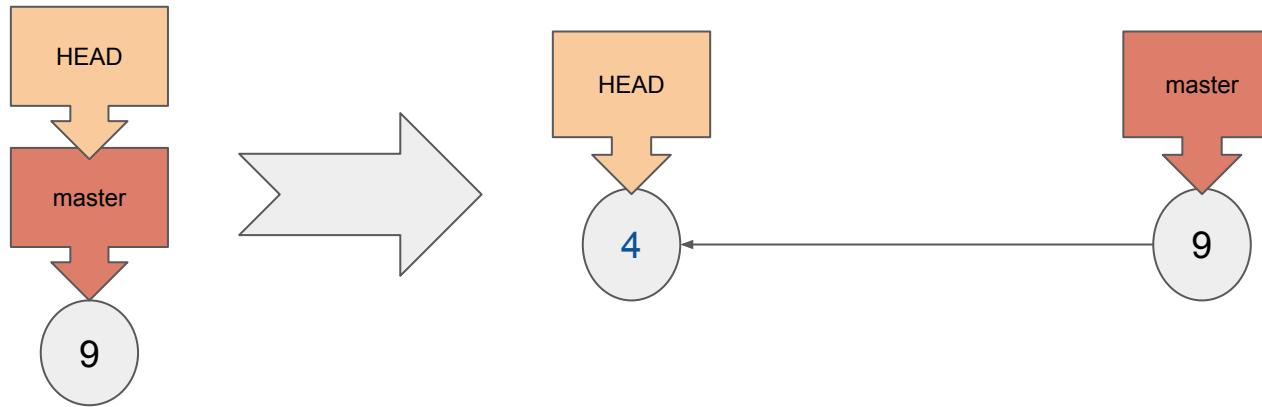
Con questo comando verrà **eliminato** il branch indicato

*Una volta eliminato il branch **non** è possibile recuperarlo, assicurarsi quindi di aver eseguito il push del branch se non si vogliono perdere le modifiche fatte.*

Checkout => Spostare HEAD

Quando Vogliamo spostarsi nella storia della repository

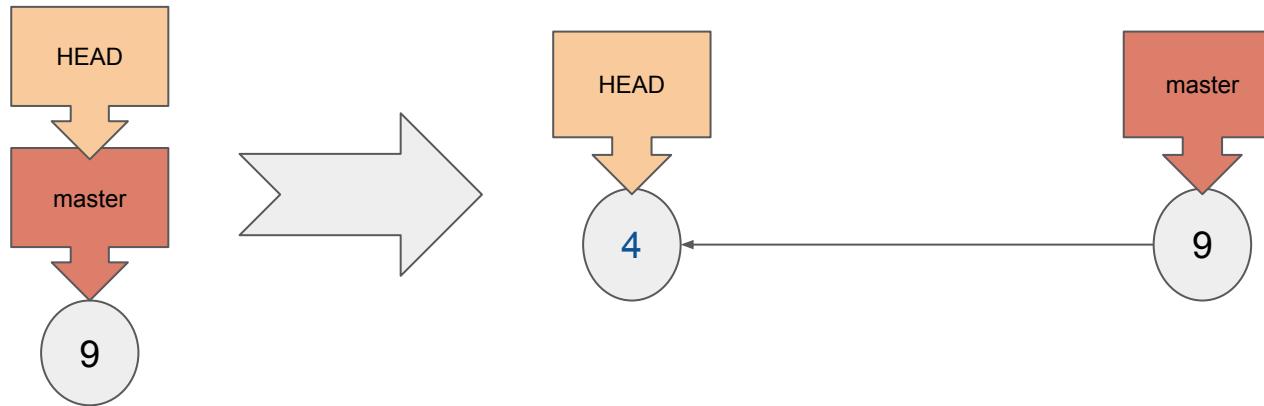
Cosa fa Sposta HEAD modificando il suo puntatore



Checkout => Sintassi

```
git checkout <sha_commit>
```

Questo comando sposta HEAD facendolo puntare a un **commit**.

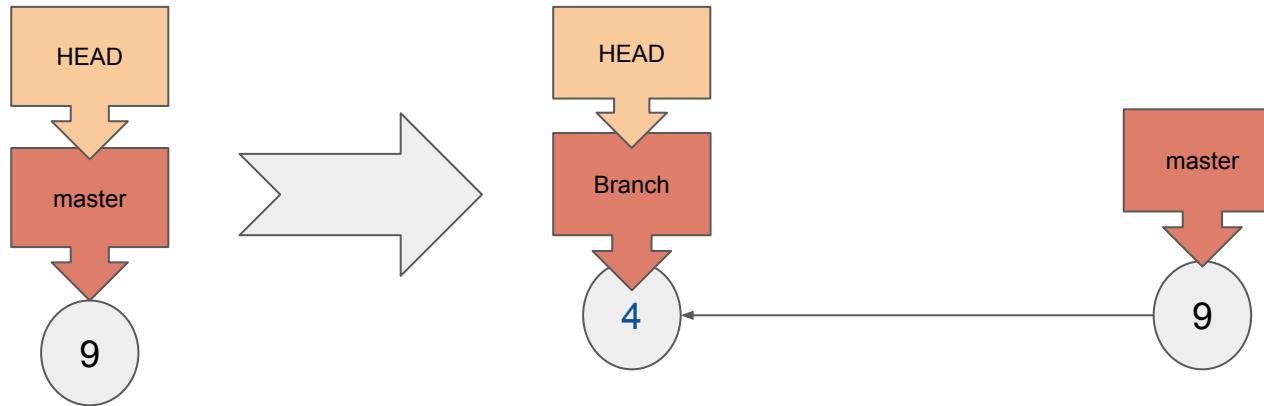


Ricordiamoci che lo stato in cui si troverà HEAD è **DETACHED**.

Checkout => Sintassi

```
git checkout <nome_branch>
```

Questo comando sposta HEAD facendolo puntare a un **branch**.



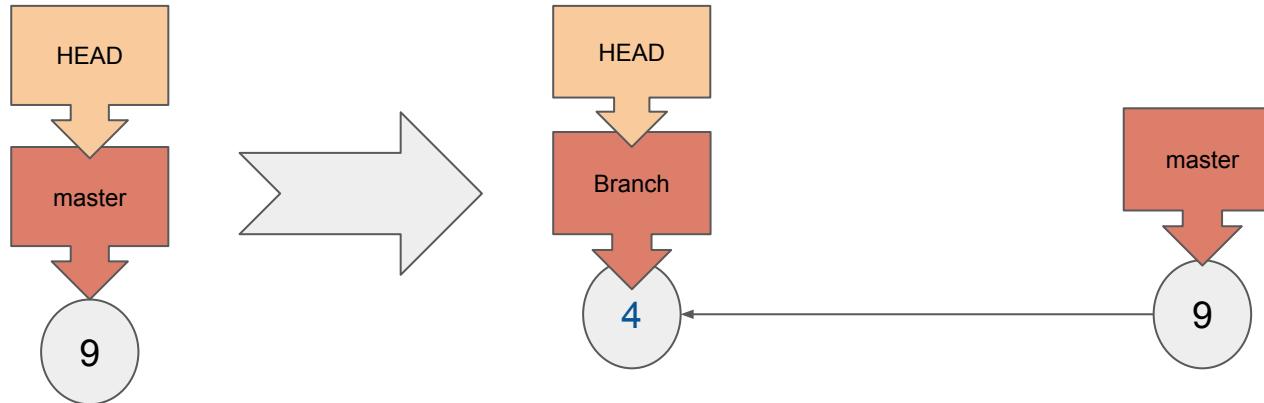
Ricordiamoci che lo stato in cui si troverà HEAD in questo caso è **Attached**.

Checkout + Branch => Sintassi

```
git branch <nome_branch> <sha_commit>  
git checkout <nome_branch>
```

```
git checkout -b <nome_branch> <sha_commit>
```

Questo comando permette di creare un nuovo branch e spostare il HEAD in un comando solo.



Ricordiamoci che lo stato in cui si troverà HEAD in questo caso è **Attached**.

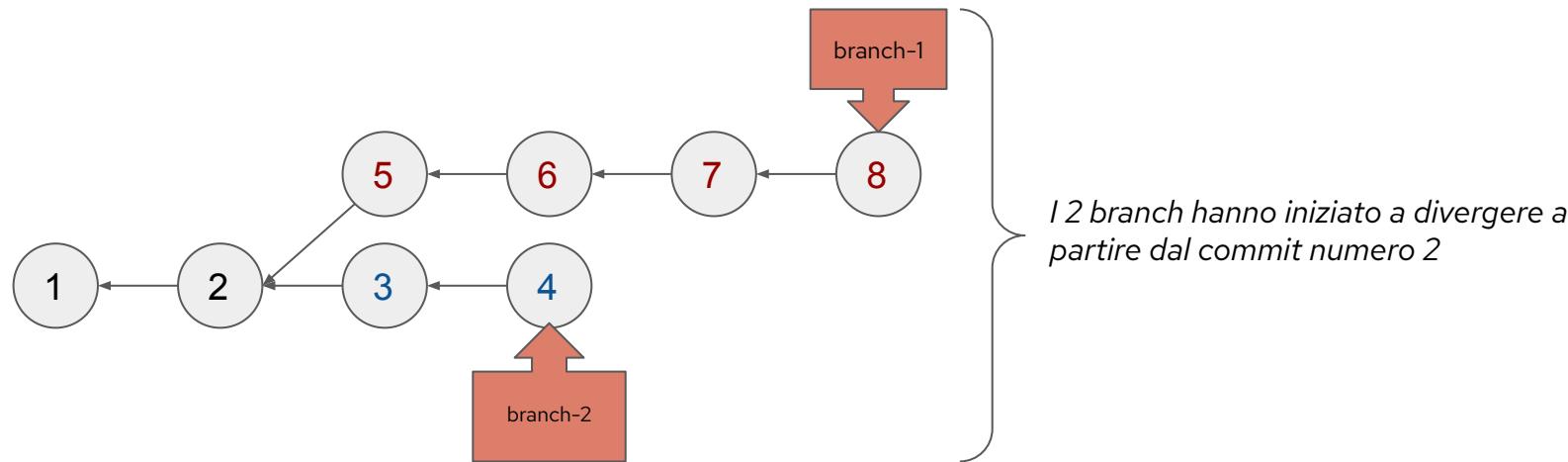
Merge => Unire branch differenti

Quando

Si vogliono importare in un branch delle modifiche fatte in un altro branch

Cosa fa

Genera un commit con tutte le modifiche effettuate a partire dal punto in cui i due branch hanno cominciato ad avere delle storie divergenti

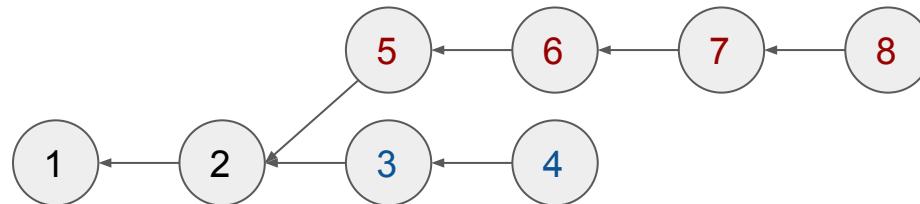


Merge => Concetti di contesto

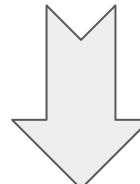
Si chiama **destinatario** del merge il commit in cui eseguiremo il merge.

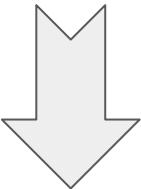
Si chiama **sorgente** del merge il commit che fa capo alle modifiche che vogliamo importare nel destinatario.

Si chiama **radice** del merge l'ultimo commit comune che hanno il destinatario e il sorgente.

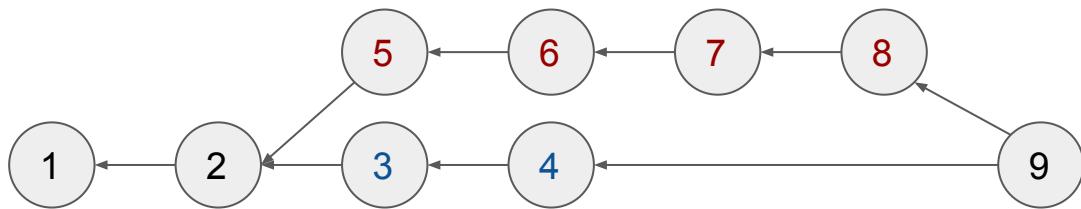


Per esempio il destinatario è il commit 4 e il sorgente è il commit 8, di conseguenza la radice è il commit 2.





L'esecuzione della merge di 8 in 4 porterà a questo risultato:



Il commit indicato con il numero 9 corrisponde al commit di merge.

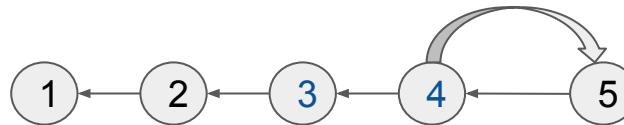
Noi non useremo i commit per riferirci alle merge, ma ai **branch** e i concetti detti in questo esempio rimangono validi.

Importante: quando eseguiamo la merge, **HEAD** si sposterà da 4 a 9.

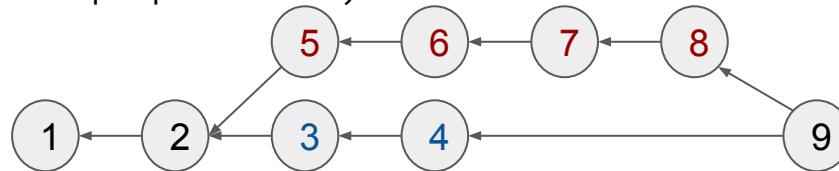
Importante 2: **HEAD** si sposterà ogni volta che viene fatto un commit.

Merge => Modalità

- Fast-forward = In questo caso il puntatore HEAD verrà spostato in avanti con la storia, come se fosse rimasto indietro nel tempo



- Recursive = Questo è il caso in cui si hanno branch divergenti tra il destinatario e il sorgente del merge (esempio precedente)



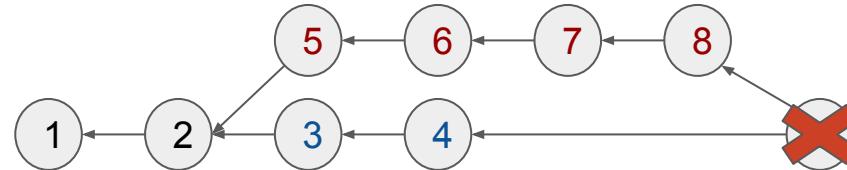
- **Reject** = vedi slide successiva

Merge => Reject

Questo è il caso in cui si hanno delle linee temporali divergenti tra il destinatario e il sorgente del merge, ma non è possibile unire ricorsivamente perchè nel destinatario è presente un commit che esegue modifiche contrapposte a un commit del sorgente

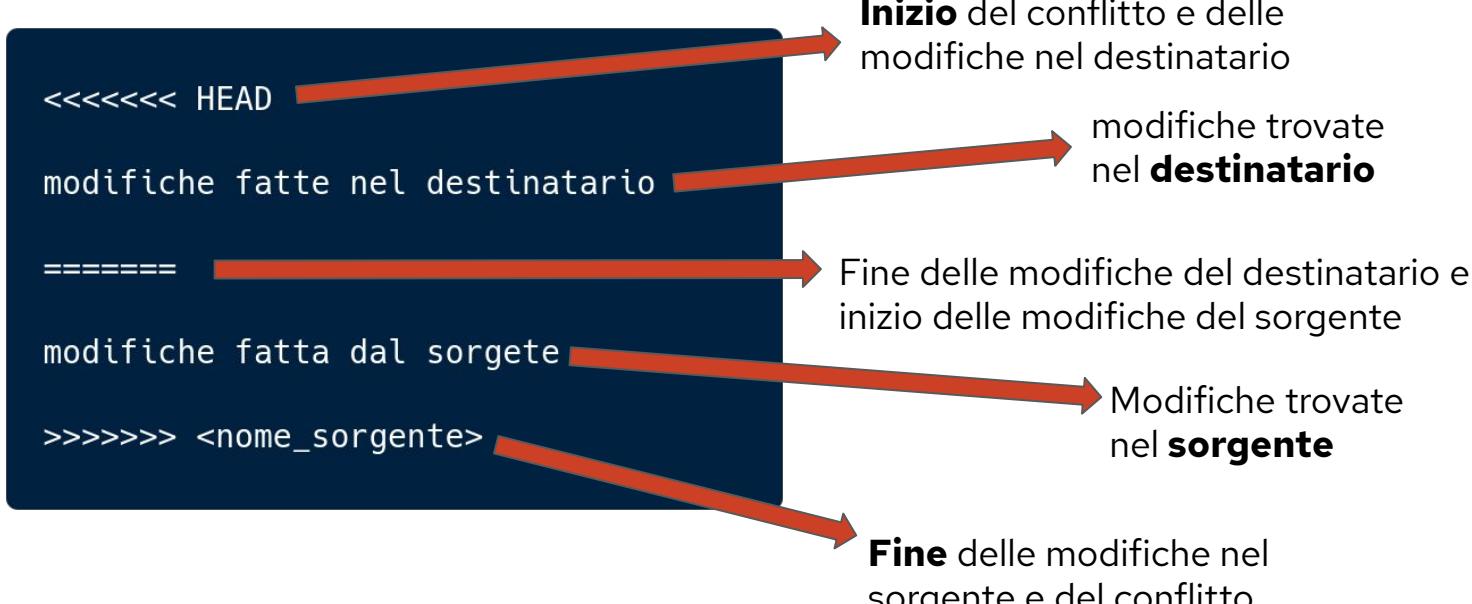
Questa tipologia è molto presente e non comporta problemi se i **conflicti** sono contenuti

In questo caso il merge viene bloccato in uno stato di attesa, in cui lo sviluppatore dovrà risolvere i conflitti



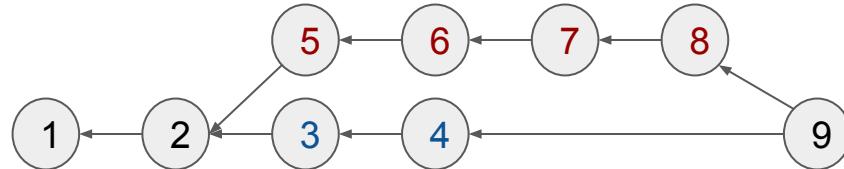
Merge => Conflitti

Se vengono trovati dei conflitti il comando merge sovrascriverà i file in conflitto:



Merge => Risoluzione conflitti

- Scegliere o unire le modifiche che portano al conflitto
- Eliminare gli indicatori del conflitto
- Marcare i file che hanno generato il conflitto come risolti, aggiungendoli alla fase stage con il comando add
- Eseguire il comando commit per generare il commit di merge.



Esempio 3

Passo 1 => Clonazione progetto remoto

1. **Eliminare** il repository contenente il repository generato con l'esempio 2
2. **Clonare** il repository remoto generato con l'esempio 2
3. **Spostarsi** all'interno del progetto

```
79% 18:29:55 ~/Workspace/accademy
> git clone git@github.com:Wabri/progetto.git
Cloning into 'progetto'...
Enter passphrase for key '/home/gab/.ssh/id_rsa':
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 7 (delta 1), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
Resolving deltas: 100% (1/1), done.

79% 18:30:05 ~/Workspace/accademy
> cd progetto/

79% 18:30:07 ~/Workspace/accademy/progetto ↵ master ✓
⌚ >
```

Passo 2 => Creazione nuovo branch

1. Visualizzare la **lista** dei branch
2. Creare un **nuovo branch** a partire dal commit precedente, usando il checkout

```
79% 18:41:05 ~/Workspace/accademy/progetto ↵ master ✓
❯ git branch
* master

79% 18:41:09 ~/Workspace/accademy/progetto ↵ master ✓
❯ git checkout -b nuovo_branch HEAD~1
Switched to a new branch 'nuovo_branch'

79% 18:42:02 ~/Workspace/accademy/progetto ↵ nuovo_branch ✓
❯
```

HEAD~1 significa il commit precedente al commit a cui punta HEAD.

Passo 3 => Nuovo file

1. Aggiungere un nuovo file **nuovo_file.txt** ed inserire **data e orario** corrente
2. Generare un nuovo commit con questa modifica

```
79% 18:46:48 ~/Workspace/accademy/progetto ↵ nuovo_branch ✓
> touch nuovo_file.txt ; date > nuovo_file.txt ; cat nuovo_file.txt

File: nuovo_file.txt
1 Thu 10 Dec 2020 06:46:58 PM CET

79% 18:46:58 ~/Workspace/accademy/progetto ↵ nuovo_branch ø1 ✘
> git add nuovo_file.txt ; git commit
[nuovo_branch f7be390] Add nuovo_file.txt
1 file changed, 1 insertion(+)
create mode 100644 nuovo_file.txt

79% 18:47:14 ~/Workspace/accademy/progetto ↵ nuovo_branch ✓
>
```

Passo 4 => Merge & Conflict

1. Spostarsi sul master
2. Eseguire il merge del nuovo branch creato nel passo 2
3. Risolvere il conflitto

```
79% 18:47:43 ~/Workspace/accademy/progetto ↵ nuovo_branch ✓
  > git checkout master ; git merge nuovo_branch
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
CONFLICT (add/add): Merge conflict in nuovo_file.txt
Auto-merging nuovo_file.txt
Automatic merge failed; fix conflicts and then commit the result.

79% 18:47:55 ~/Workspace/accademy/progetto ↵ master +1 •1 ✘
⑧1 ⌂ > vim nuovo_file.txt

79% 18:48:25 ~/Workspace/accademy/progetto ↵ master +1 •1 ✘
⑧ ⌂ > git add nuovo_file.txt ; git commit
[master db92294] Merge branch 'nuovo_branch'

79% 18:48:35 ~/Workspace/accademy/progetto ↵ master ↑2 ✓
⑧ ⌂
```

Passo 5 => Clean & Push

1. Una volta che siamo sicuri che il merge sia andato a buon fine possiamo eliminare il branch creato nel passo 2
2. Eseguire il push delle modifiche aggiornando il remote

```
79% 18:49:02 ~/Workspace/accademy/progetto ⌂ master ↑2 ✓
⌚ > git branch -d nuovo_branch
Deleted branch nuovo_branch (was f7be390).

79% 18:49:08 ~/Workspace/accademy/progetto ⌂ master ↑2 ✓
⌚ > git push origin master
Enter passphrase for key '/home/gab/.ssh/id_rsa':
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 630 bytes | 315.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To github.com:Wabri/progetto.git
    ad26ebc..db92294  master → master

79% 18:49:56 ~/Workspace/accademy/progetto ⌂ master ✓
⌚ >
```

Branching Workflow

Branching Workflow

- Gestione dei branch
- Multiple remotes
- Fetch
- Rebase
- Rebase vs Merge

Branch => Management

Abbiamo già visto il funzionamento del comando branch per visualizzare i branch localmente presenti, crearne di nuovi e eliminarli (slide 74).

Con questo comando è anche possibile:

1. Ottenere l'ultimo commit fatto per ogni branch
2. Recuperare la lista dei branch che sono già mergiati, non mergiati o i branch che sono in remote diversi
3. Rinominare un branch
4. Modificare il branch di riferimento in upstream

Branch => Management

```
git branch --verbose
```

Restituirà i branch locali con lo sha e la descrizione dell'**ultimo commit**

```
git branch --merged  
git branch --no-merged
```

Questi due comandi restituiranno una lista dei branch che sono stati **uniti o no al branch** in cui siamo posizionati

```
git branch --move vecchio-name nuovo-name
```

L'attributo move consente di **modificare il nome** di un branch

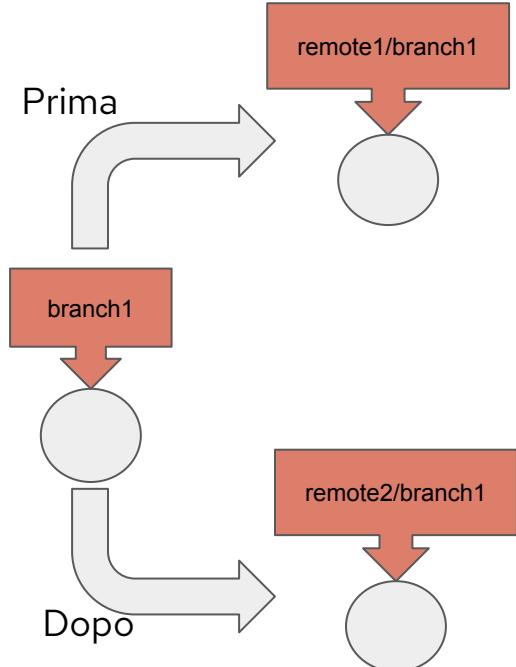
Branch => Upstream

```
git branch --set-upstream=remote/branch
```

Il comando permette di impostare il remote a cui deve far riferimento il branch locale.

Nell'esempio di destra è stato modificato il branch upstream di branch1 che da remote1/branch1 diventa remote2/branch2, il comando per attuare questa modifica è il seguente:

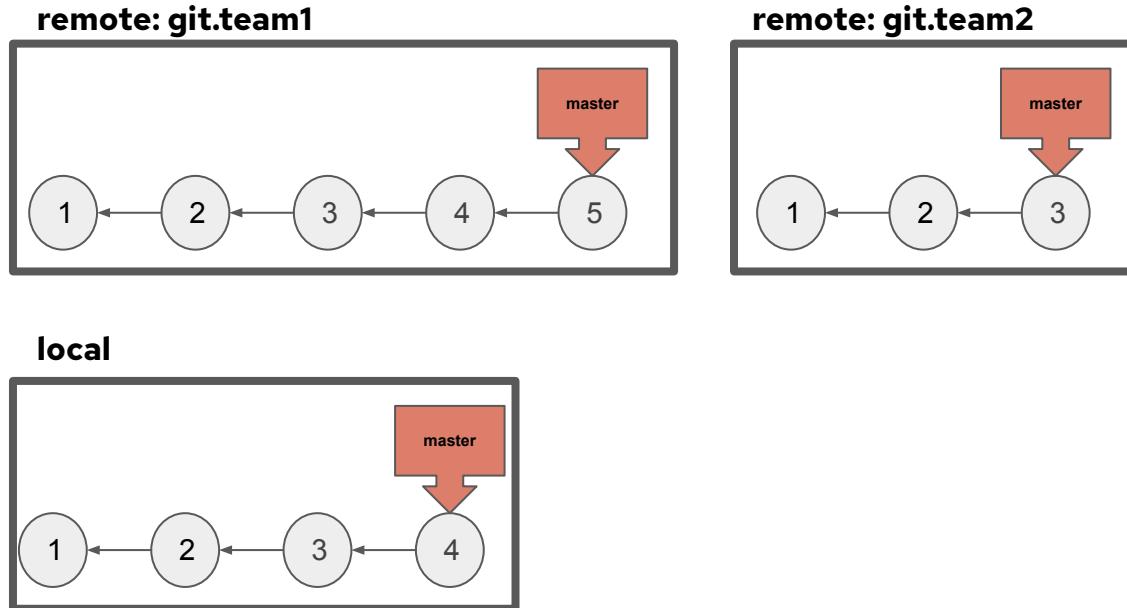
```
git branch --set-upstream=remote2/branch1 branch1
```



Remote => multiple remotes

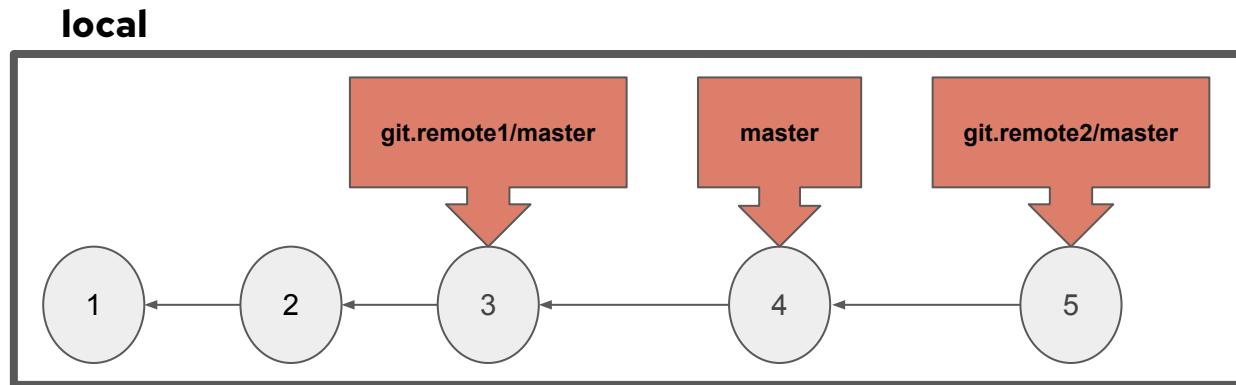
Come già detto nella slide 49 i remote di un progetto possono essere più di uno.

Potremmo avere una situazione di questo tipo:



Remote => multiple remotes

Se entrambi i remote sono stati aggiunti nella repository locale allora abbiamo 3 diversi branch chiamati master, ma con remote diversi:



Fetch => Aggiornare remote locali

Quando Vogliamo aggiornare i remote del repository **locale**

Cosa fa **Scarica le modifiche nel remote**



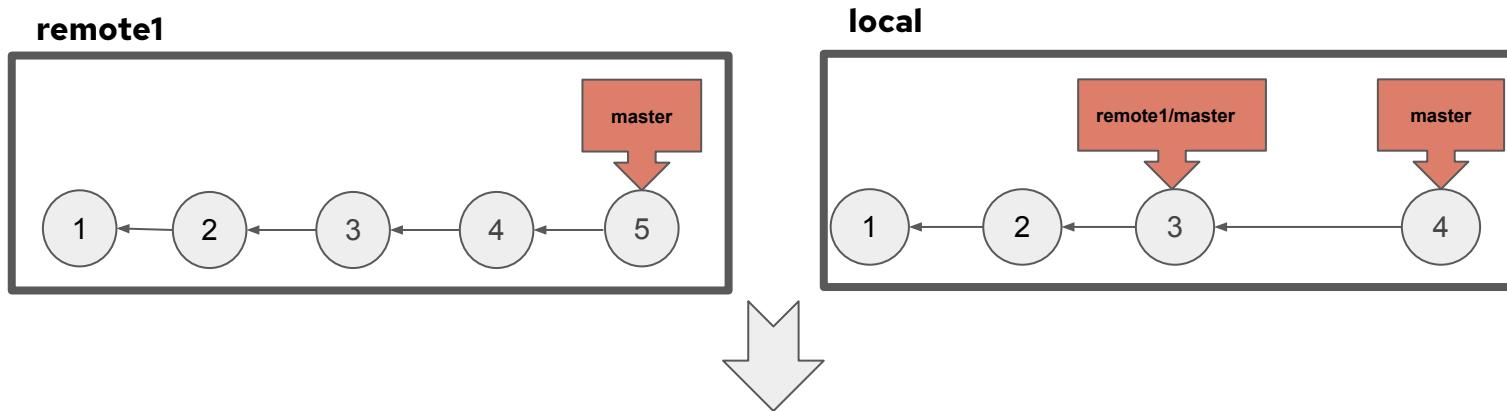
Notiamo che questo comando è molto simile al comando `pull`, in effetti il `pull` è un macro comando che esegue prima una `fetch` e poi una `merge`.

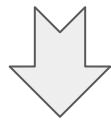
Fetch => Sintassi

```
git fetch <nome_remoto>
```

Questo comando permette di scaricare i commit dal remoto indicato aggiornando di conseguenza i riferimenti locali.

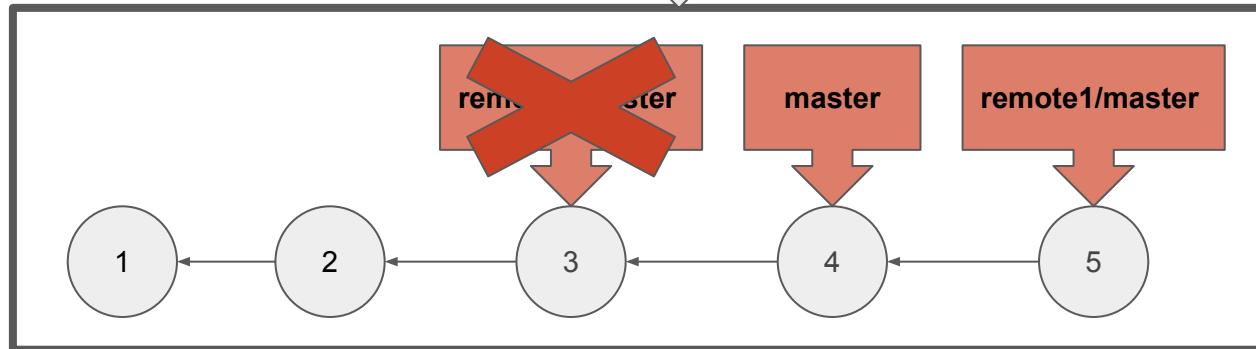
Non influisce in alcun modo con i branch locali.





```
git fetch remotel
```

local



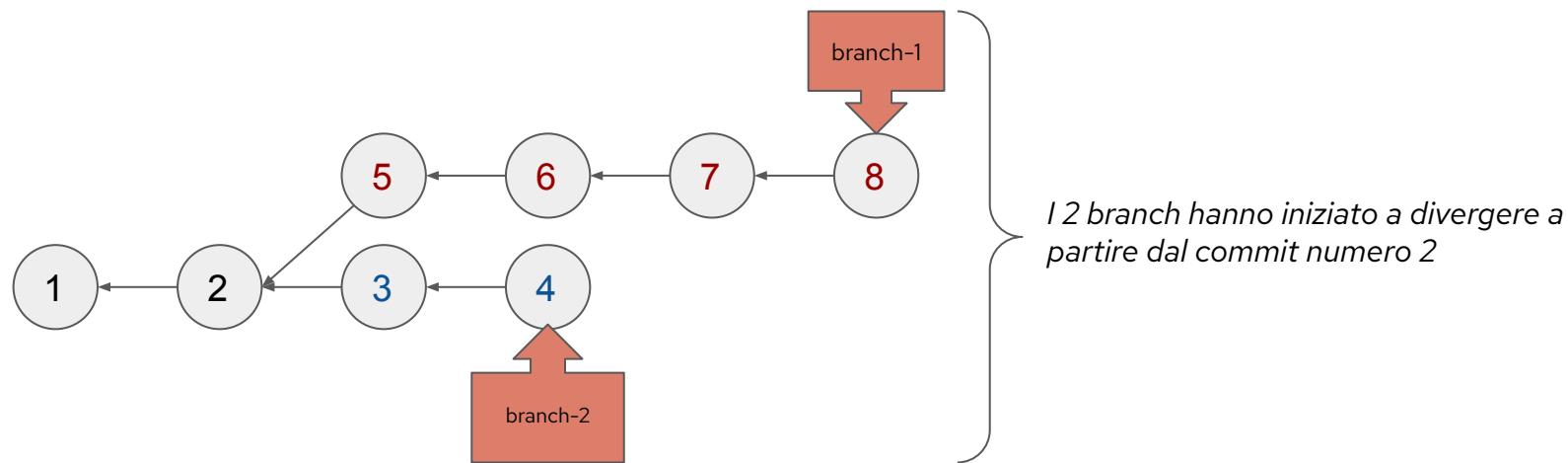
Rebase => Allineare branch differenti

Quando

Si vogliono importare in un branch delle modifiche fatte in un altro branch

Cosa fa

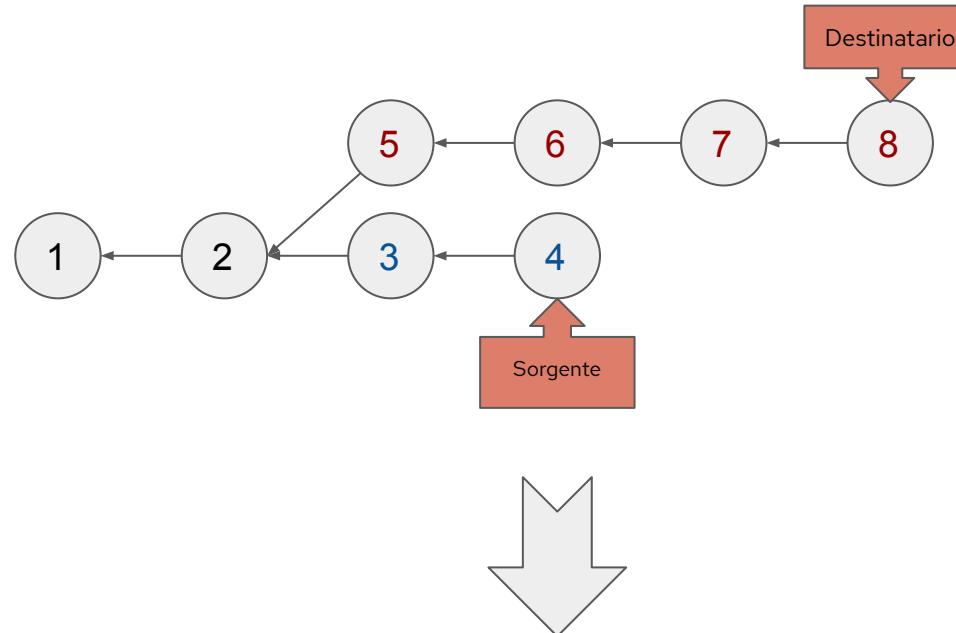
Allinea il branch destinatario con il branch sorgente a partire dal primo commit divergente.



Rebase => Concetti di contesto

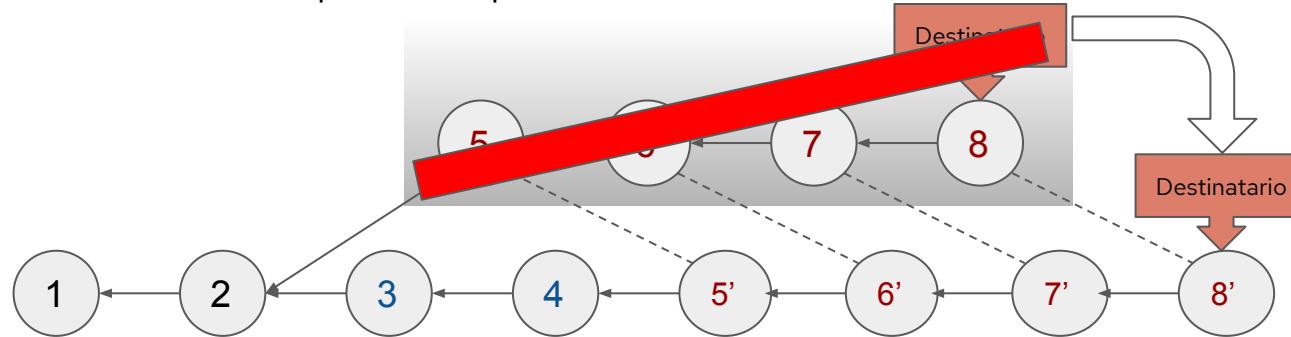
Si chiama **destinatario** del rebase il branch in cui eseguiremo il rebase.

Si chiama **sorgente** del rebase il branch con cui vogliamo allineare il destinatario.





L'esecuzione della rebase porterà a questo risultato:



A partire dal commit 4 vengono **applicati uno per uno i commit** del destinatario includendo quindi le modifiche fatte dai commit del destinatario nel branch sorgente.

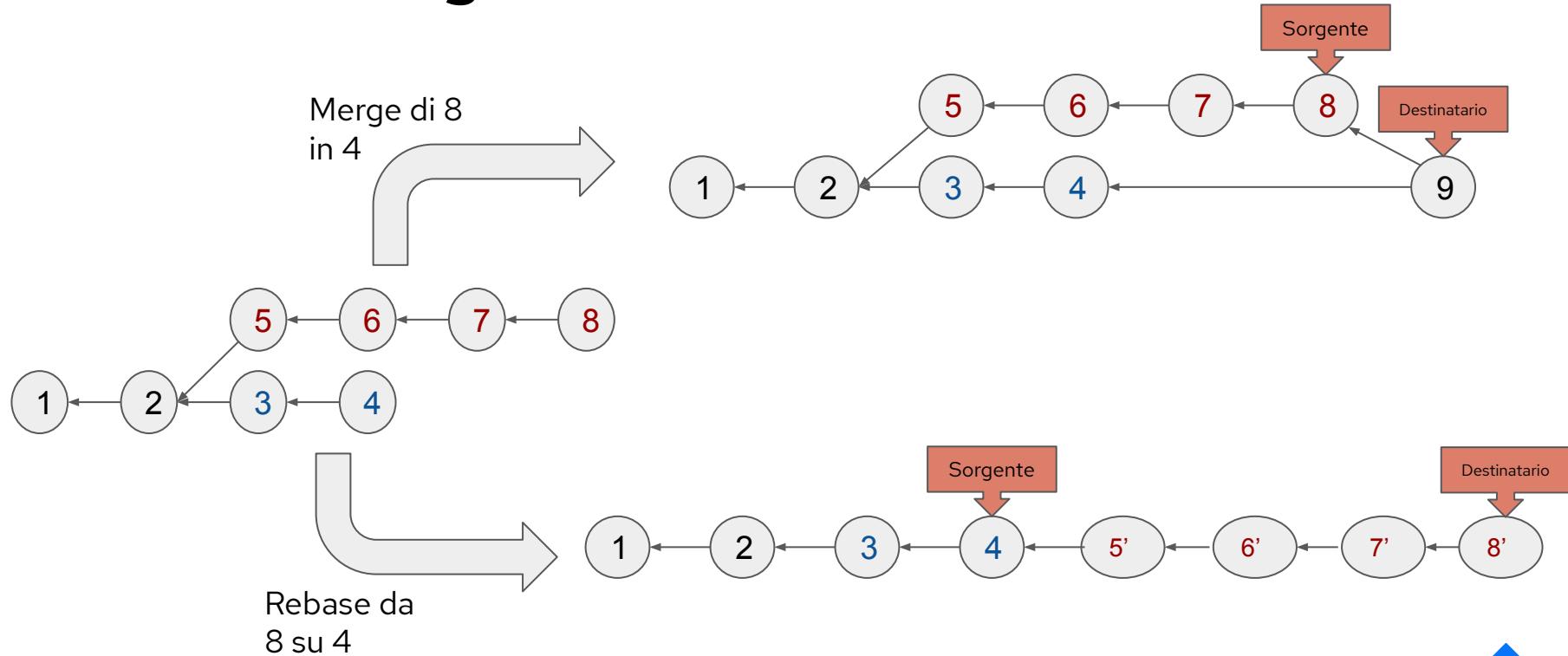
Fintanto che il rebase non è concluso il progetto rimane in uno **stato pendente**.

Come durante un merge è possibile che l'applicazione di un commit del destinatario **provochi un conflitto**, che deve essere risolto e accettato.

Importante: Il branch sorgente **non** viene in alcun modo modificato.

Importante 2: ogni commit che viene applicato avrà uno **sha diverso**.

Rebase vs Merge



Altri comandi utili

Altri comandi utili

- diff
- log
- blame
- cherry-pick
- reset
- restore
- revert

Diff => Visualizzare i cambiamenti

Quando

Dobbiamo controllare quali sono i cambiamenti tra un commit e l'altro, o se ci sono modifiche che ancora non sono state aggiunte alla fase di staged.

Cosa fa

Elenca le modifiche file per file

Sintassi

```
git diff
```

```
git diff <sha_commit>
```

```
git diff --name-only
```

```
git diff --staged
```

Log => Visualizzare i commit

Quando

Vogliamo avere una lista di tutti i commit fatti con la descrizione completa

Cosa fa

Crea una lista di commit fatti nel branch dove eseguiamo il comando

Sintassi

```
git log
```

```
git log --all
```

```
git log --graph --all --oneline
```

```
git log --graph
```

```
git log --graph --all
```

Blame => autore della modifica

Quando

Vogliamo controllare chi ha fatto delle modifiche a un file o a una specifica linea di codice.

Cosa fa

Restituisce riga per riga: lo sha del commit, l'autore, quando è stata fatta la modifica e il numero di riga.

Sintassi

```
git blame <nome_file>
```

```
git blame <nome_file> --show-email
```

Cherry-pick => applicare un commit

Quando

Vogliamo importare le modifiche fatte in un commit specifico

Cosa fa

Applica un commit con tutte le modifiche relative al commit indicato

Sintassi

```
git cherry-pick <sha_commit>
```

```
git cherry-pick --edit <sha_commit>
```

```
git cherry-pick <sha_commit1>...<sha_commit2>
```

Reset => Modificare HEAD

Quando

Vogliamo modificare la posizione del puntatore HEAD

Cosa fa

Modifica il commit a cui punta il puntatore HEAD

Sintassi

```
git reset <branch>
```

```
git reset --soft
```

```
git reset --hard
```

```
git reset --mixed
```

ATTENZIONE

Se HEAD si trova su un branch, l'esecuzione di **reset in modalità mixed o hard** modificherà la posizione del branch quindi attenzione.

Restore => Ripristinare file modificati

Quando

Vogliamo ripristinare lo **stato clean** di file che si trovano in stato dirty

Cosa fa

Elimina tutte le modifiche fatte su file che si trovano in stato dirty

Sintassi

```
git restore <percorso_file>
```

Revert => Uncommit

Quando

Vogliamo eliminare le modifiche apportate da un commit

Cosa fa

Ripristina lo stato di un processo prima di modifiche fatte da uno specifico commit

Sintassi

```
git revert <sha_commit>
```

```
git revert <sha_commit1>...<sha_commit2>
```

Git Workflow

Git Workflow

Esistono delle raccomandazioni per l'uso in contesti di alto livello per rendere lo sviluppo e la manutenzione di un progetto git organizzata.

Git è lo strumento per il lavoro di sviluppo software in team e più il team è grande più è necessario avere uno standard per l'organizzazione della repository.

Esistono diverse correnti di pensiero che corrispondono a diversi git workflow:

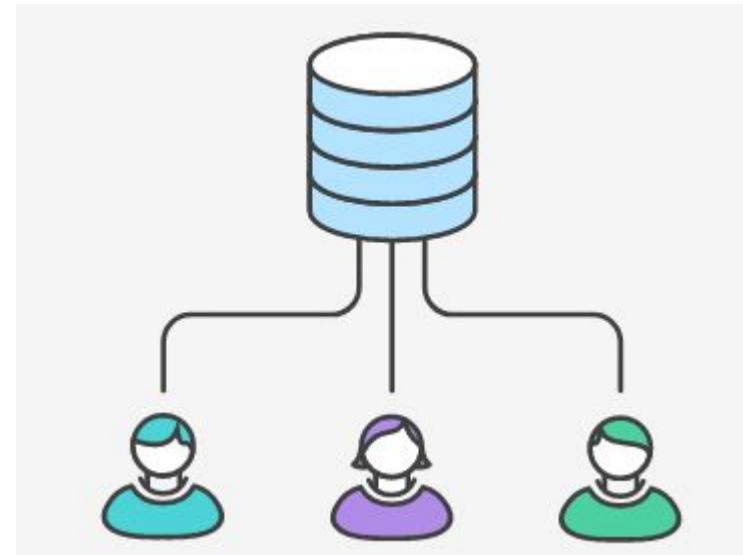
- Centralized => di norma usato durante la transizione da un vcs di tipo centralizzato
- Feature branching => estensione logica del precedente
- Gitflow => Pensato 10 anni fa, facilita e implementa le pratiche di devops
- Forking => Ogni sviluppatore ha il proprio repository remoto

Centralized

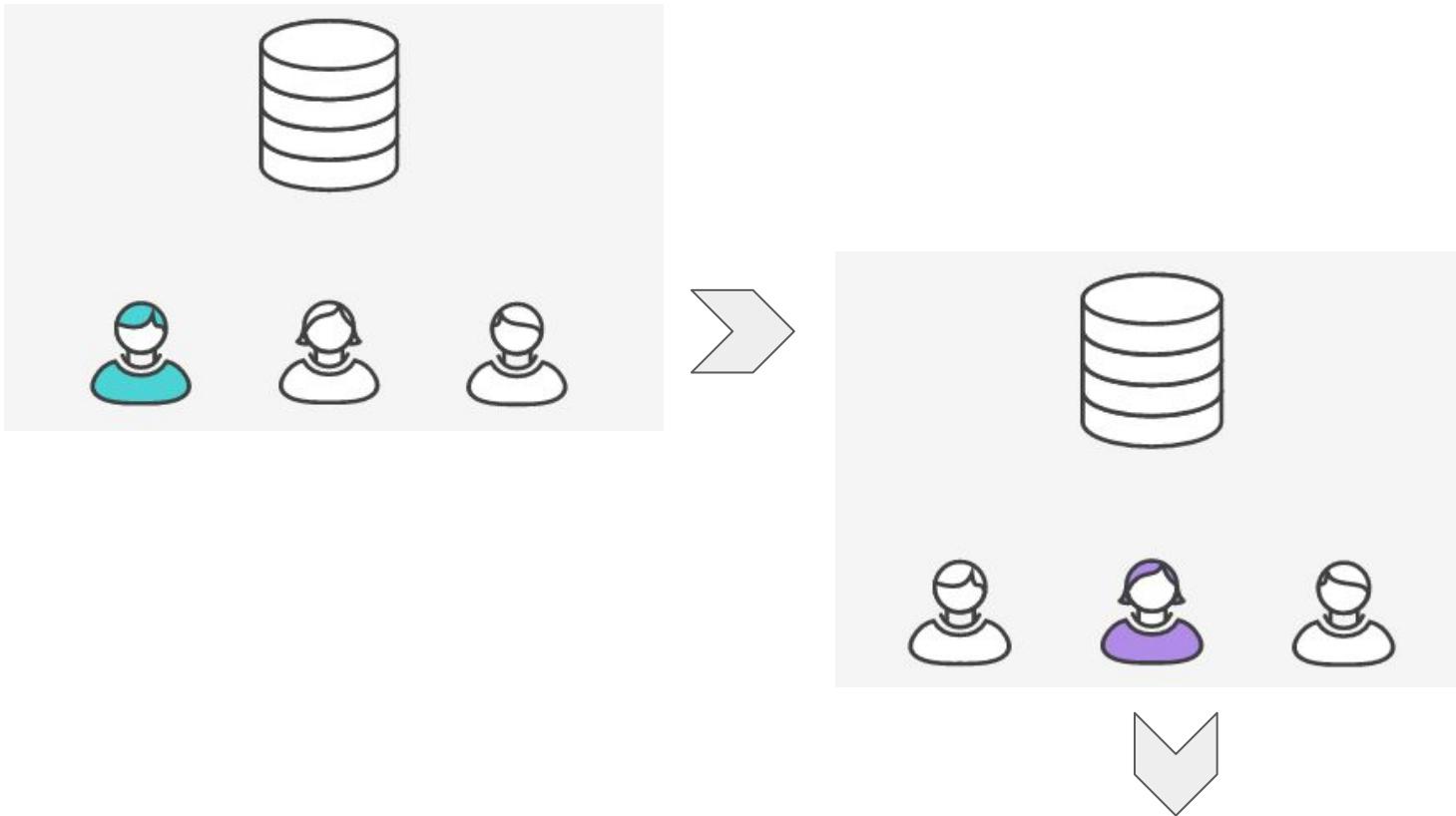
Tutti usano un branch, non esiste nessuna forma di branching, né di complesse forme di merge e rebase.

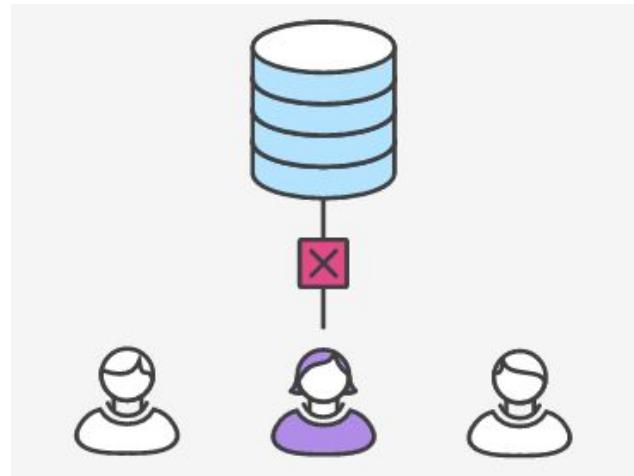
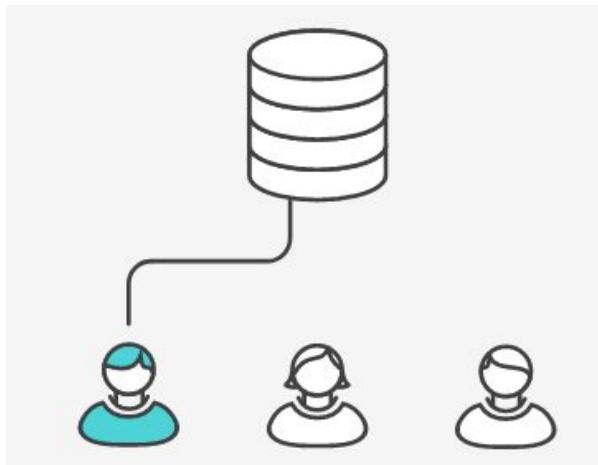
Molta probabilità che avvengano dei conflitti.

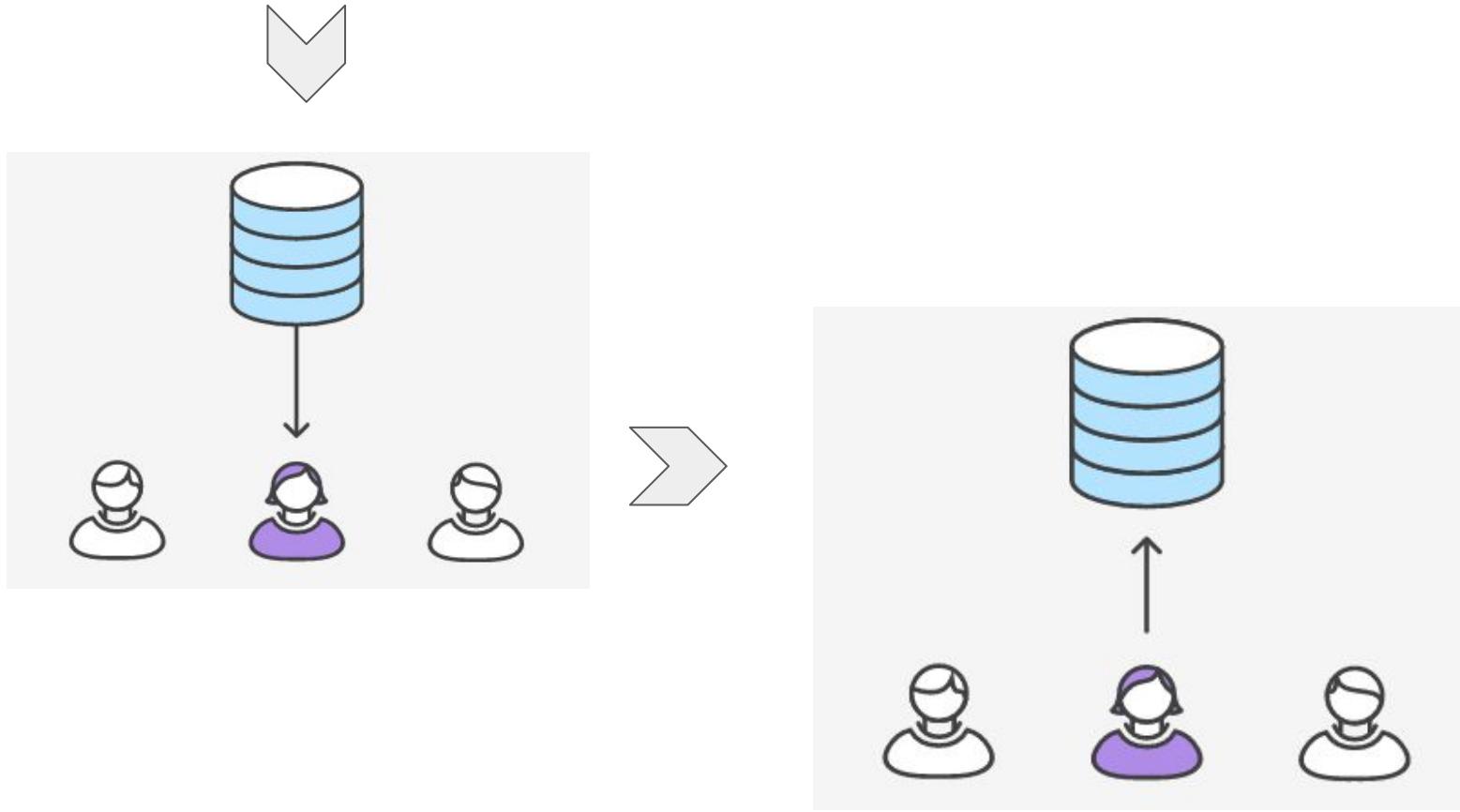
Semplice da usare, ma elevata generazione di errori.



Centralized => Esempio





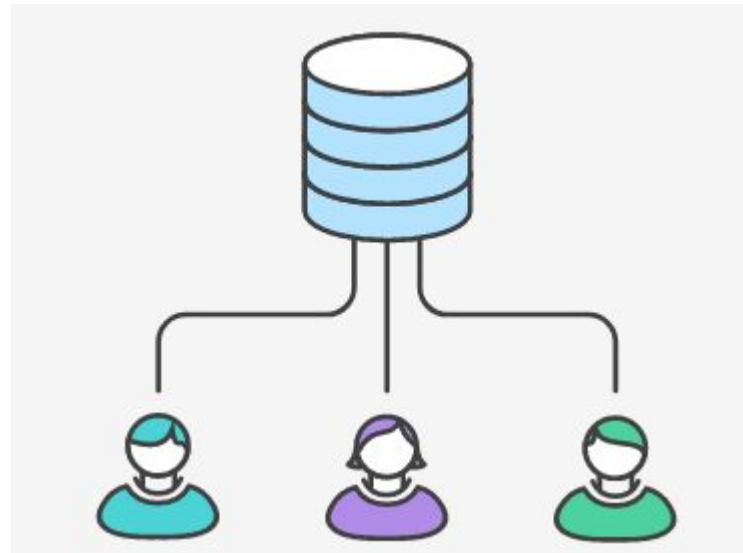


Feature Branching

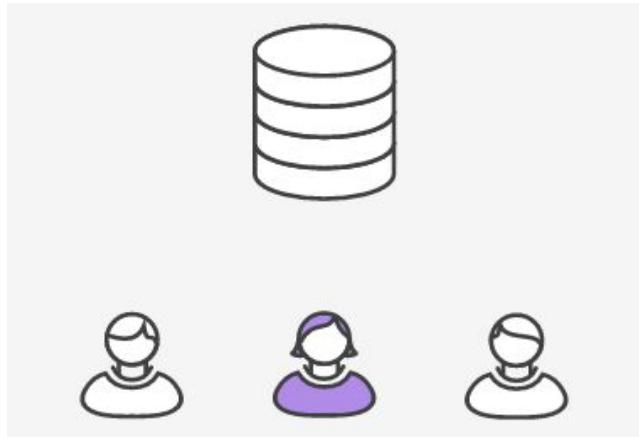
Evoluzione del central workflow.

Anche in questo caso abbiamo un branch principale, ma ogni nuova modifica deve prendere piede in un branch di sviluppo specifico.

Ogni nuova modifica deve iniziare a partire dal branch principale.



Feature Branching => Esempio



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



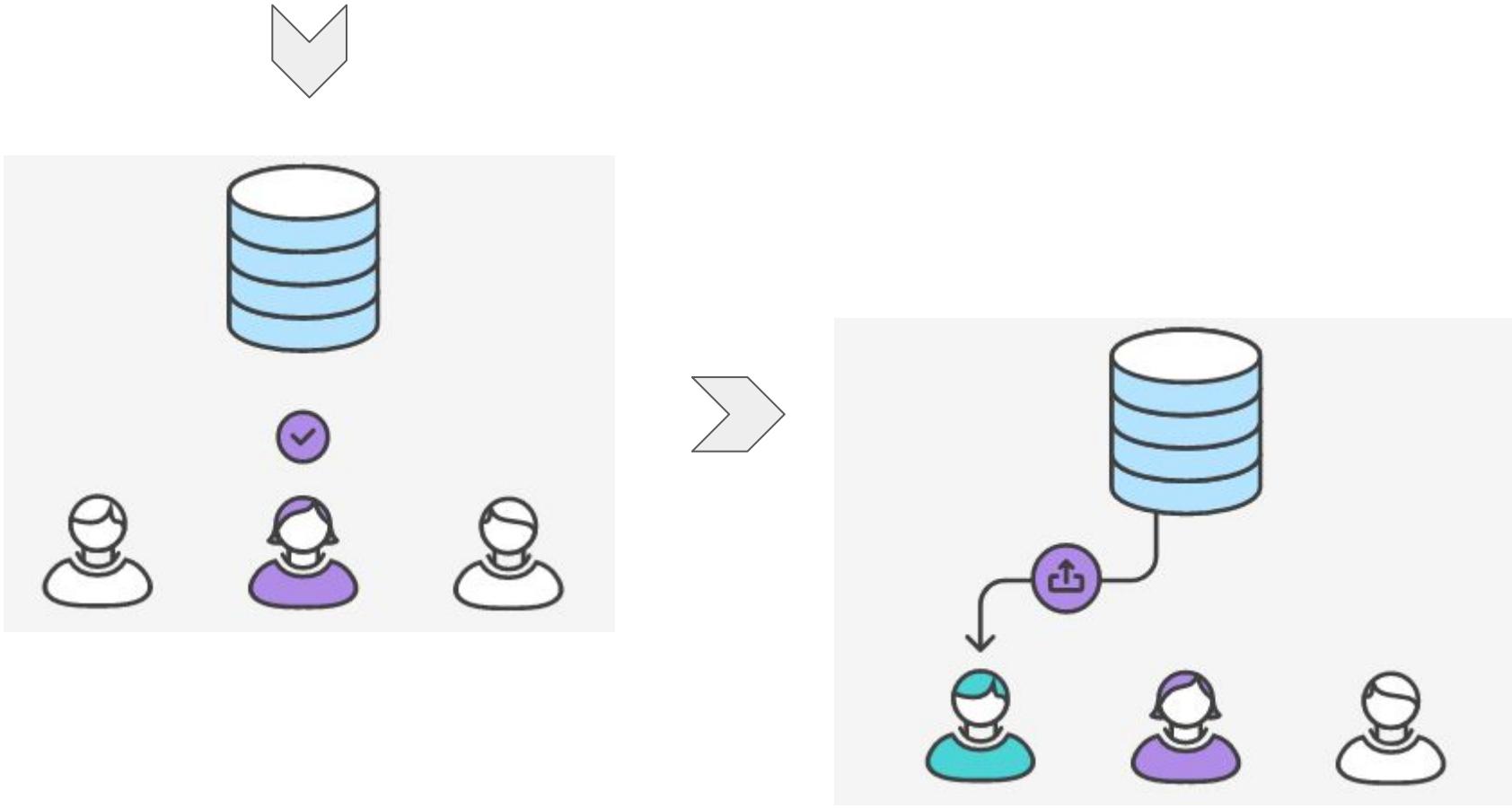


```
git checkout -b new-feature
```



```
git add <files>  
git commit  
git push -u origin new-feature
```





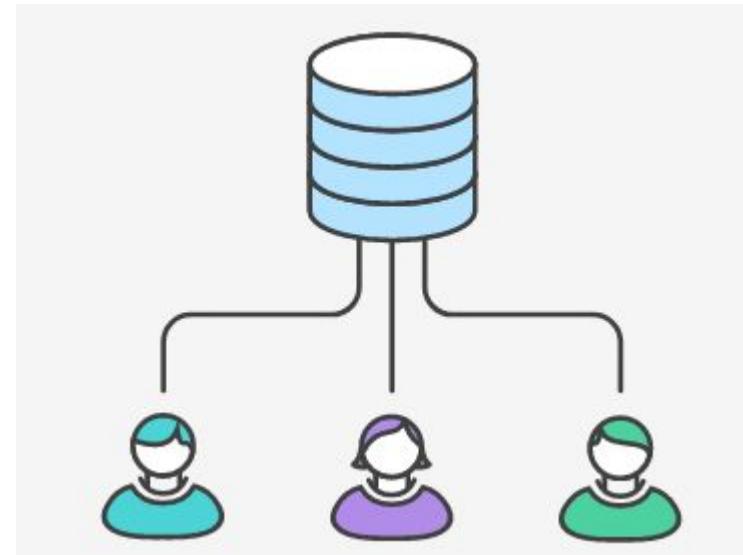
GitFlow

GitFlow

Prima versione nel 2010 con un post nel blog di Vincent Driessen che è possibile ancora trovare <https://nvie.com/posts/a-successful-git-branching-model/>

Pensato per progetti e team di grandi dimensioni in cui sono presenti pratiche di devops, ma utilizzabile anche altrimenti.

Si basa su 3 concetti principali di branch: protected, feature, hotfix e release.

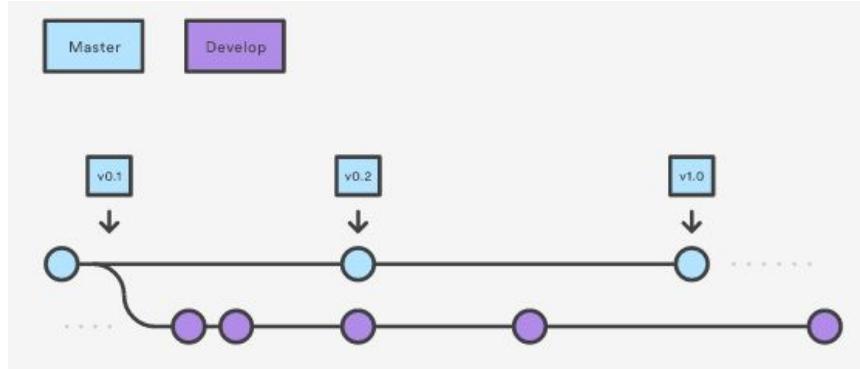


Protected Branch

A differenza di altri workflow questo utilizza 2 branch per tenere traccia della storia del progetto:

- Master/Main = in cui sono contenute le versioni ufficiali
- Develop = usato per integrare i branch di feature

Ogni nuovo commit generato su master viene taggato con la versione relativa al software.

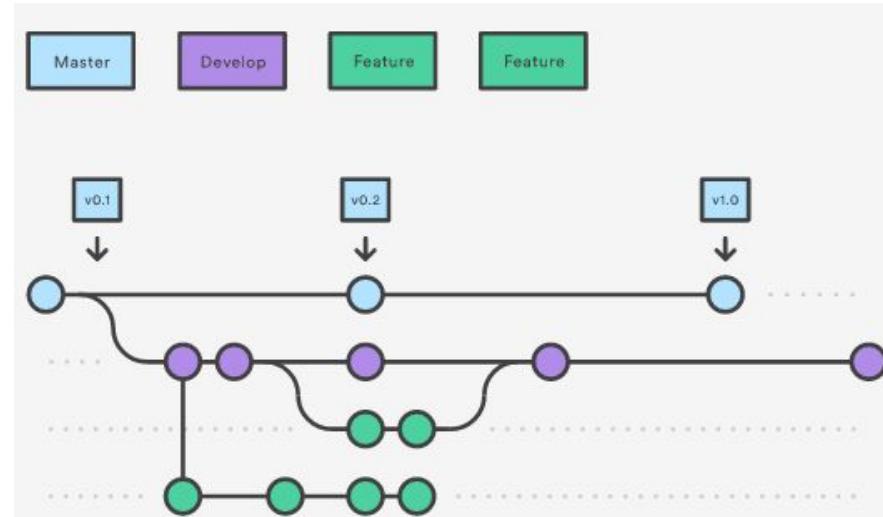


Feature Branch

Il concetto riprende le definizioni date nello workflow omonimo, quindi ogni nuova feature deve essere sviluppata nel suo specifico branch.

La differenza è che deve partire dal branch develop e non dal master.

Quando viene completata la feature il branch viene unito al develop e mai deve accadere che un branch di questo tipo interagisca con il master direttamente.



Hotfix e Release => generare versioni

La principale differenza dai normali feature branch è che questi servono per generare nuove release del software, quindi servono per creare nuovi commit su master.

Hotfix sono branch usati per fixare velocemente problemi o bug che possono presentarsi. Inizia a partire da un commit su master, lo risolve e poi viene generata una nuova versione eseguendo un merge di questo branch su master.

Release sono branch usati per preparare il codice per una versione ufficiali del software. Iniziano a partire dal branch develop, viene preparato il codice a una nuova versione (per esempio modificando file di configurazione o integrato del codice) e poi viene generata una versione eseguendo un merge di questo branch su master.

GitFlow => installazione

[nvie / gitflow](#)

Code Issues Pull requests Actions Projects Wiki Security Insights

develop 4 branches 6 tags Go to file Code

jeromebaum Use git_do where appropriate

2 15aab26 on Sep 25, 2012 401 commits

contrib	make the installer for msysgit search in ~/bin as well	9 years ago
shFlags @ 2fb06af	Include shFlags repo as a Git submodule.	11 years ago
.gitignore	Add debian/ubuntu packaging.	10 years ago
.gitmodules	Revert "Change submodule URL from git:// to http://"	11 years ago
AUTHORS	Credit where credit is due.	9 years ago
Changes.mdwn	Merge branch 'debian-installer' into develop	9 years ago
LICENSE	Whitespace issue.	11 years ago
Makefile	Replace contrib/debian/ with a note on how to install on Debian.	10 years ago
README.mdwn	Fix broken link to original blog-post	9 years ago

About
Git extensions to provide high-level repository operations for Vincent Driessens's branching model.

[nvie.com/posts/a-successful-git-br...](#)

Readme View license

Releases
6 tags

Packages

<https://github.com/nvie/gitflow>

GitFlow => Esempio

```
git checkout develop  
git checkout -b feature_branch
```



```
git checkout develop  
git merge feature_branch
```



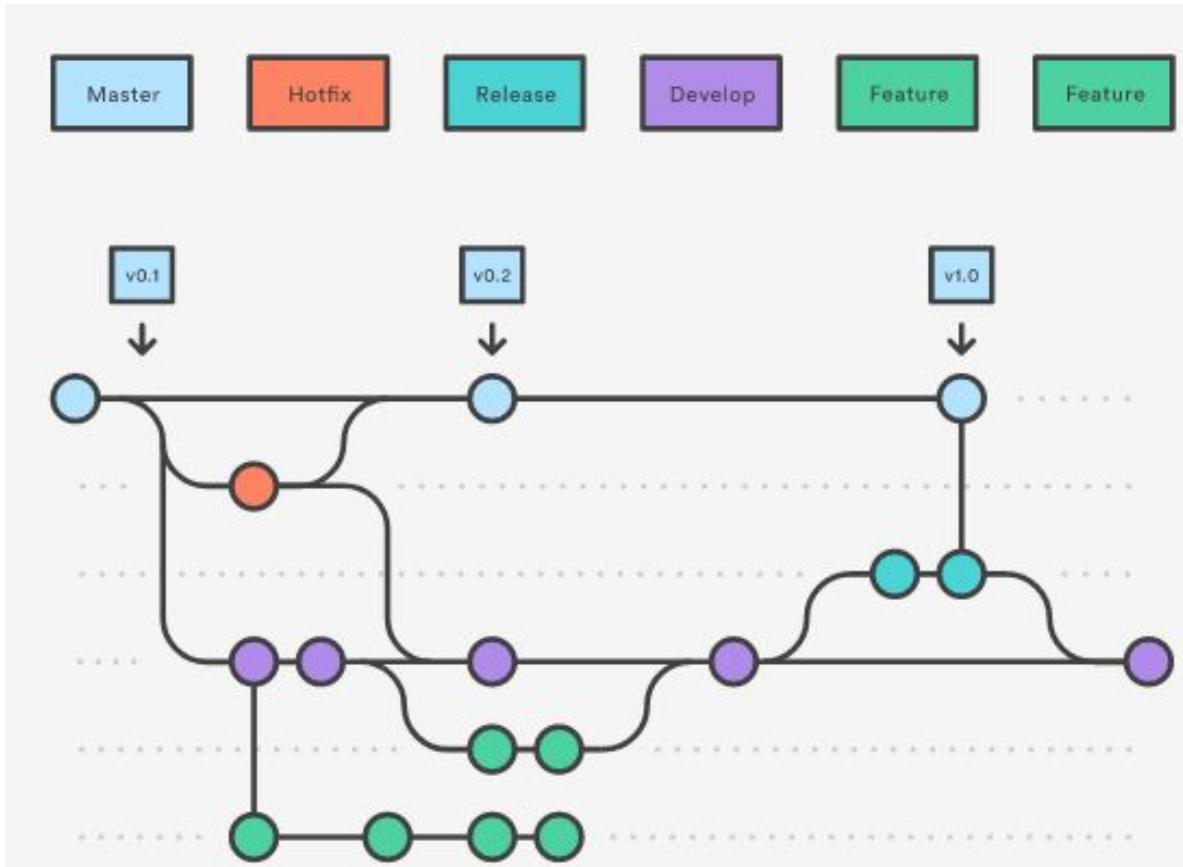
```
git flow feature start feature_branch
```



```
git flow feature finish feature_branch
```



GitFlow => Schema finale



Letture consigliate

Letture consigliate

- ProGit - <https://git-scm.com/book/en/v2>
- Prima versione del gitflow -
<https://nvie.com/posts/a-successful-git-branching-model/>
- Workflow - <https://www.atlassian.com/git/tutorials/comparing-workflows>
- Ogni comando ha una help page completa frutto di 15 anni di lavoro, di oltre 2 mila sviluppatori di tutto il mondo e di oltre 800 versioni