



# git

# Introduzione a GIT

TECNICHE PER LA PROGETTAZIONE E LO SVILUPPO DI  
APPLICAZIONI INFORMATICHE - JUNIOR FULL STACK WEB  
DEVELOPER

Mastroianni Davide  
[davide.mastroianni@designcoaching.net](mailto:davide.mastroianni@designcoaching.net)

# 1. CONTROLLO DI VERSIONE

---

# CONTROLLO DI VERSIONE

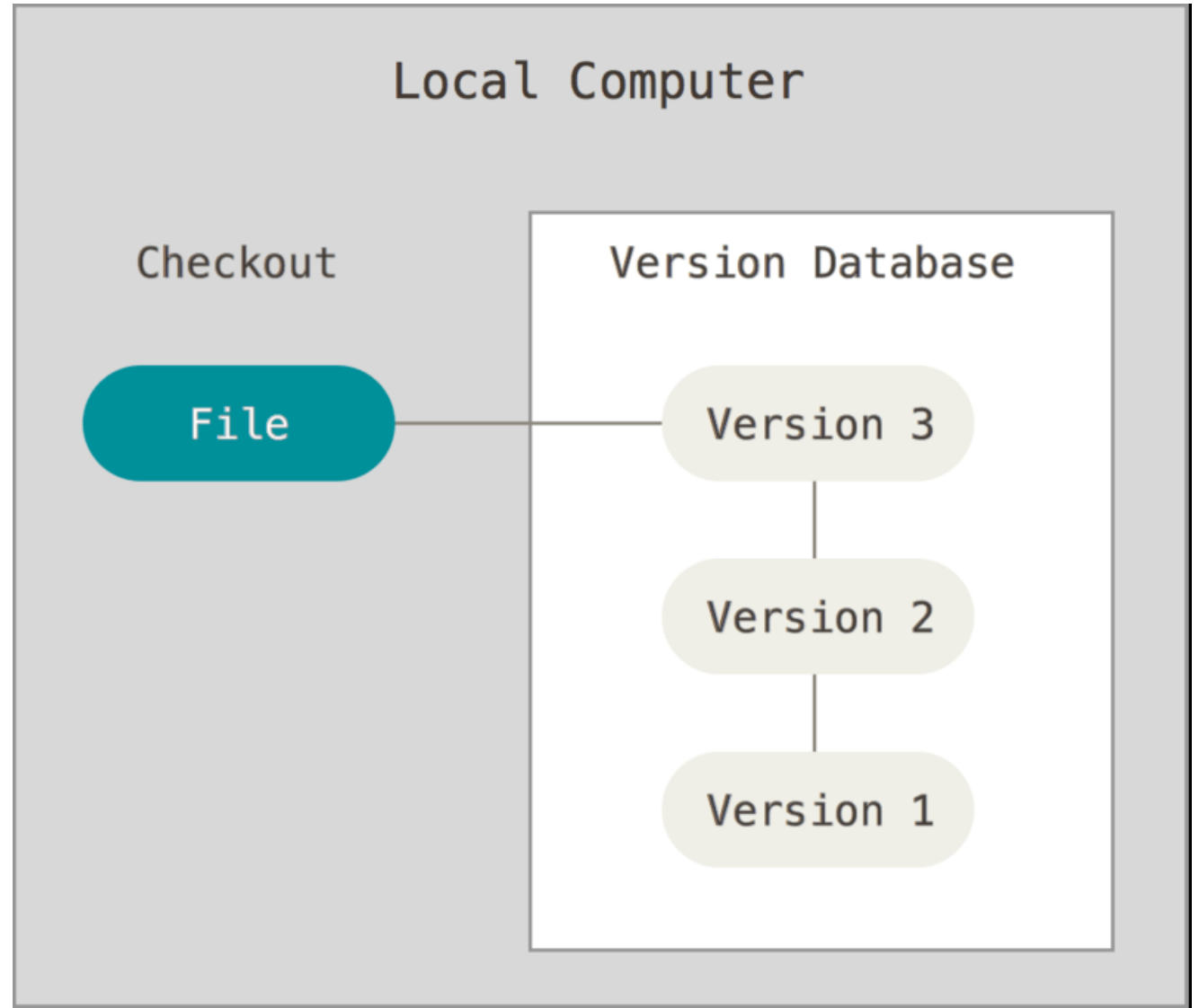
---

- Il controllo di versione è un sistema che registra, nel tempo, i cambiamenti ad un file o ad una serie di file.
- Ogni tipo di file può essere sottoposto al controllo di versione.
- Il sistema che si occupa di registrare la storia di un file o ripristinare un file ad uno stato precedente è definito "Sistema per il controllo di versione" o "Version Control System" (VCS).

# CONTROLLO DI VERSIONE

Sistema di Controllo di Versione Locale (VCS)

Gestito localmente sulla singola macchina.



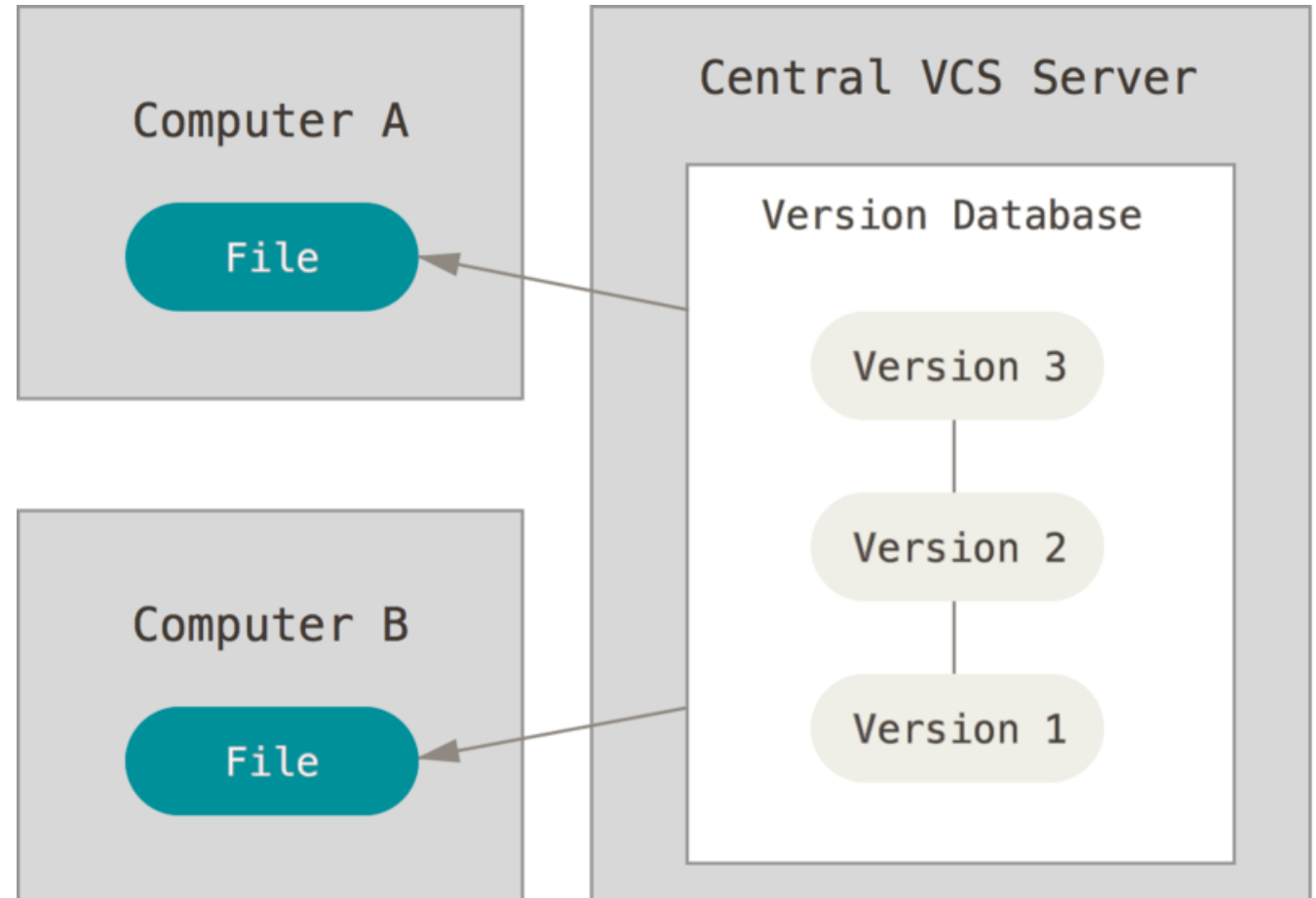
# CONTROLLO DI VERSIONE

- Sistema di Controllo di Versione Centralizzato (CVCS)

Gestito in remoto su un server centralizzato, favorendo la collaborazione. Solo il server mantiene la storia della repository.

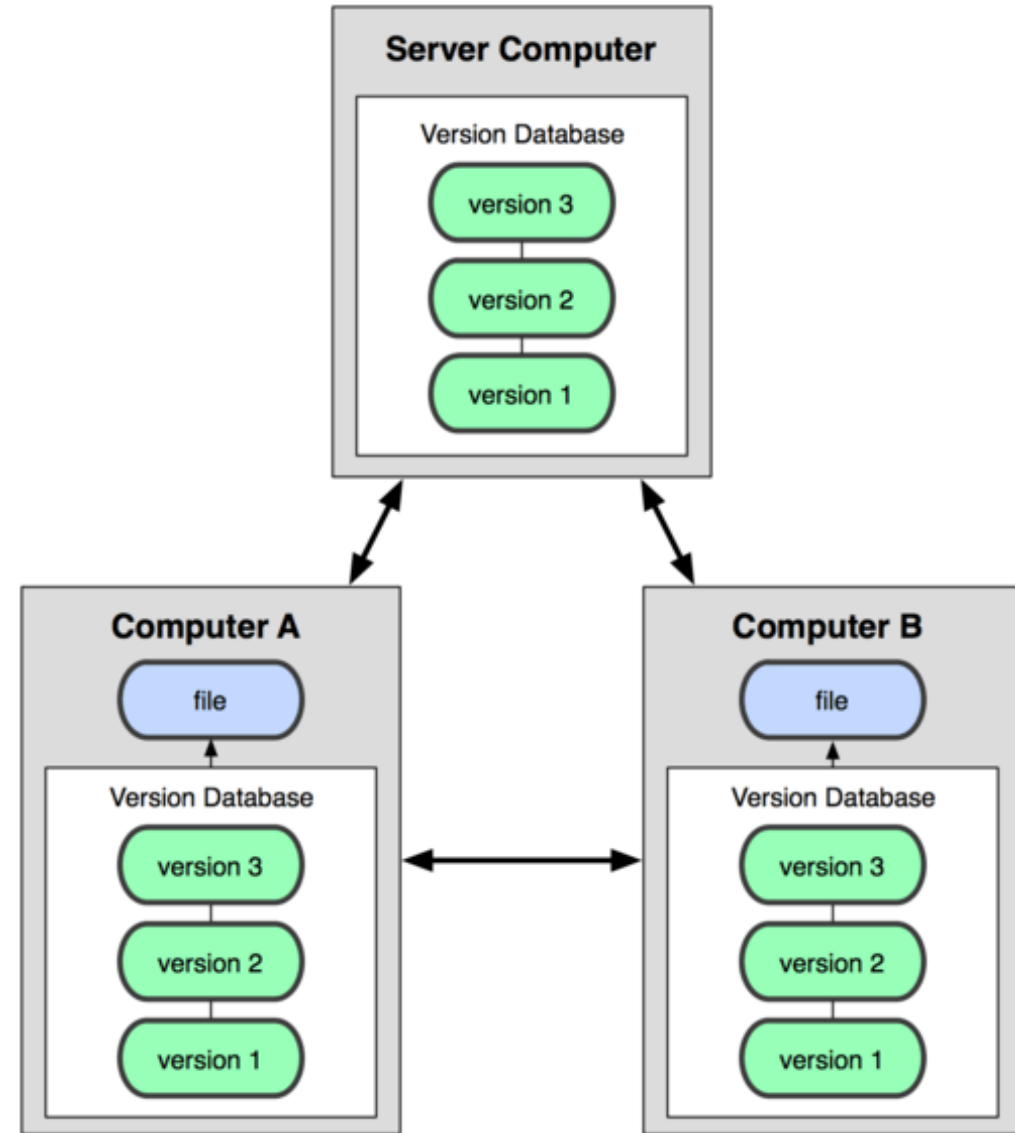
Le modifiche locali sul computer non sono versionate.

Sistemi come Subversion (svn), CVS e Preforce sono VCS di tipo centralizzato



# CONTROLLO DI VERSIONE

- Sistema di Controllo di Versione Distribuito (DVCS)  
Ogni repository locale è la copia esatta della repository remota.  
C'è una sola repository principale, a cui si fa riferimento per sincronizzare tutte le repository distribuite.

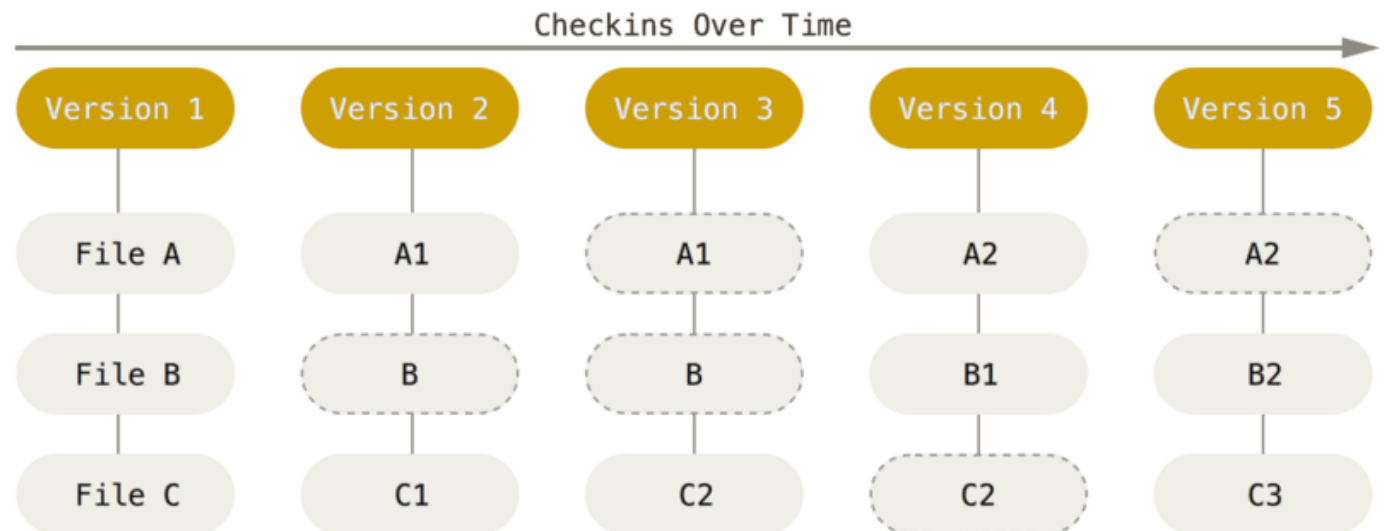
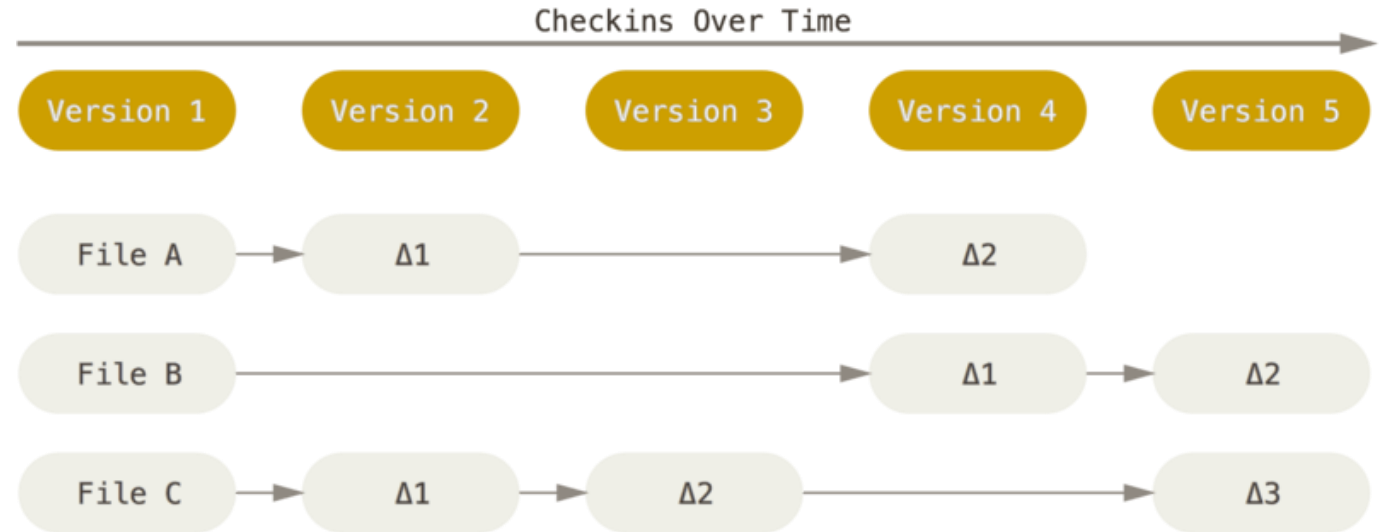


# GESTIONE DELLE VERSIONI

La maggiore differenza tra GIT e gli altri VCS è il modo con il quale GIT memorizza i suoi dati.

Normalmente i VCS salvano le informazioni come liste di cambiamenti di file.

GIT, invece, vede i dati del progetto come un'istantanea del progetto in un certo istante (**snapshot**). Git pensa ai suoi dati come uno "stream of snapshots"



## 2. LE BASI DI GIT

---



# CONFIGURAZIONE

- git config
- git config --global user.name "nomeUtente"
- git config --global user.email "utente@mail.com"
- git config --list

```
MINGW64/c/Users/davide
davide@davide-pc MINGW64 ~
$ git config
usage: git config [<options>]

Config file location
  --global          use global config file
  --system          use system config file
  --local           use repository config file
  --worktree        use per-worktree config file
  -f, --file <file> use given config file
  --blob <blob-id>  read config from given blob object

Action
  --get             get value: name [value-regex]
  --get-all        get all values: key [value-regex]
  --get-regexp      get values for regexp: name-regex [value-regex]
  --get-urlmatch    get value specific for the URL: section[.var] URL
  --replace-all    replace all matching variables: name value [value_regex]
  --add            add a new variable: name value
  --unset          remove a variable: name [value-regex]
  --unset-all     remove all matches: name [value-regex]
  --rename-section  rename section: old-name new-name
  --remove-section remove a section: name
  -l, --list       list all
  -e, --edit       open an editor
  --get-color      find the color configured: slot [default]
  --get-colorbool  find the color setting: slot [stdout-is-tty]

Type
  -t, --type <>   value is given this type
  --bool          value is "true" or "false"
  --int           value is decimal number
  --bool-or-int   value is --bool or --int
  --path          value is a path (file or directory name)
  --expiry-date   value is an expiry date

Other
  -z, --null      terminate values with NUL byte
  --name-only     show variable names only
  --includes      respect include directives on lookup
  --show-origin   show origin of config (file, standard input, blob, command line)
  --default <value> with --get, use default value when missing entry

davide@davide-pc MINGW64 ~
$ |
```

# REPOSITORY GIT

---

Esistono due modi per creare un nuovo repository GIT per un progetto:

- Convertire la cartella esistente del progetto in un progetto GIT, utilizzando il comando **git init**
- Clonare un progetto esistente da un'altra repository, **git clone <url>**

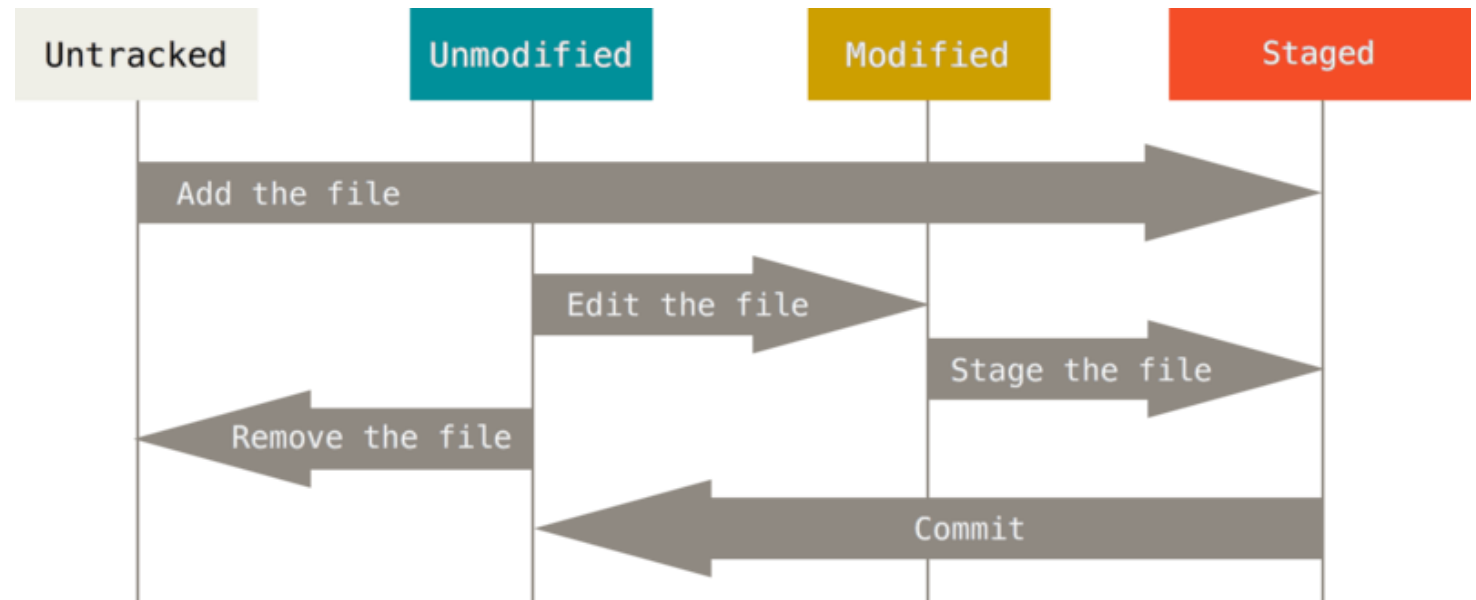
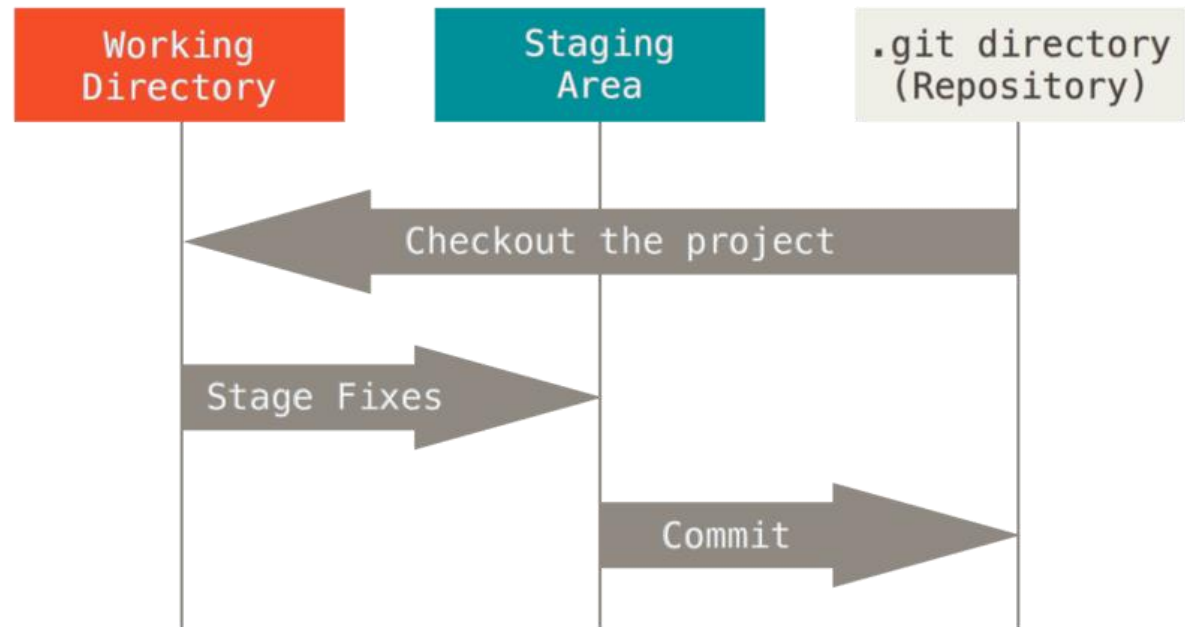
# CICLO DI VITA DEI FILE

I file sono contenute in tre diverse aree:

- Working Directory
- Staging Area
- .git repository

Inoltre possono avere quattro stati differenti:

- Untracked
- Unmodified
- Modified
- Staged



# COMANDI BASE

- git status
- git add nomeFile.ext  
O  
git add regExp
- git commit -m "commento"

```
davide@t1700: ~/Desktop/work/corso-git
davide@t1700:~/Desktop/work/corso-git$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
davide@t1700:~/Desktop/work/corso-git$ git add link.html
davide@t1700:~/Desktop/work/corso-git$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   link.html
davide@t1700:~/Desktop/work/corso-git$ git commit -m "aggiunto link.html al repository"
[master (root-commit) 497293b] aggiunto link.html al repository
 1 file changed, 12 insertions(+)
 create mode 100644 link.html
davide@t1700:~/Desktop/work/corso-git$ git status
On branch master
nothing to commit, working directory clean
davide@t1700:~/Desktop/work/corso-git$
```

# GIT LOG

Il comando **git log** mostra una serie di informazioni sui commit passati, visualizzando dal più recente al più vecchio.

Tra le informazioni si trovano:

- Il **checksum**, che identifica il commit.
- L'**autore** del commit.
- La **data** e l'**ora** del commit.
- Il **messaggio** del commit.

```
davide@t1700: ~/Desktop/work/corso-git
davide@t1700:~/Desktop/work/corso-git$ git log
commit 497293b018110e3ecbcf3b11ca704a8906c01b5f
Author: davide <davide.mastroianni@designcoaching.net>
Date:   Mon Dec 17 14:25:15 2018 +0100

    aggiunto link.html al repository
davide@t1700:~/Desktop/work/corso-git$ git log --oneline --decorate --graph --all
* 497293b (HEAD -> master) aggiunto link.html al repository
davide@t1700:~/Desktop/work/corso-git$
```

# GIT DIFF

Per vedere quali file nella repository sono stati modificati, ma non sono ancora stati inseriti nell'area di staging si utilizza il comando **git diff**. Questo comando confronta cosa c'è nella tua cartella di lavoro con quello che c'è nella tua area di staging.

Se, invece, si vuole vedere cosa c'è nell'area di staging che farà parte del prossimo commit si utilizza **git diff --staged**

```
davide@t1700: ~/Desktop/work/corso-git
davide@t1700:~/Desktop/work/corso-git$ vim link.html
davide@t1700:~/Desktop/work/corso-git$ git diff
diff --git a/link.html b/link.html
index 54be735..f4d071c 100644
--- a/link.html
+++ b/link.html
@@ -3,7 +3,9 @@
<body>

<h2>HTML Links</h2>
-<p>HTML links are defined with the a tag:</p>
+<p>Questo paragrafo è stato modificato</p>
+
+<h3>Un sottotitolo a caso</h3>

<a href="https://www.youtube.com/watch?v=dQw4w9WgXcQ">This is a link</a>

davide@t1700:~/Desktop/work/corso-git$ git add link.html
davide@t1700:~/Desktop/work/corso-git$ git diff
davide@t1700:~/Desktop/work/corso-git$ git diff --staged
diff --git a/link.html b/link.html
index 54be735..f4d071c 100644
--- a/link.html
+++ b/link.html
@@ -3,7 +3,9 @@
<body>

<h2>HTML Links</h2>
-<p>HTML links are defined with the a tag:</p>
+<p>Questo paragrafo è stato modificato</p>
+
+<h3>Un sottotitolo a caso</h3>

<a href="https://www.youtube.com/watch?v=dQw4w9WgXcQ">This is a link</a>

davide@t1700:~/Desktop/work/corso-git$
```

# MODIFICA DEI FILE CON GIT

---

Per rimuovere un file dalla repository git si utilizza il comando

- **git rm nomeFile.ext**

In tal modo il file selezionato viene rimosso dalla repository e dal disco. Se si vuole mantenere il file sul disco ma rimuoverlo dalla repository è necessario utilizzare il flag `cached` assieme al comando.

- **git rm --cached nomeFile.ext**

Per rinominare un file versionato esiste un apposito comando git.

- **git mv "nomeFileVecchio.ext" "nomeFileNuovo.ext"**

Modificando il nome di un file attraverso IDE, o semplicemente da interfaccia utente, verrà interpretato come aver eliminato il file ed averne creato uno nuovo.

NB: per rendere tutte le modifiche effettive è necessario aggiungerle al commit successivo.

# GIT UNDO

---

- **git reset HEAD nomeFile.ext** -> rimuove dalla staging area il file specificato. Non vengono perse le modifiche.
- **git checkout -- nomeFile.ext** -> rimuove dalla staging area il file specificato e lo ripristina allo stato dell'ultimo commit. Tutte le modifiche al file vengono perse
- **git commit --amend** -> ripristina l'ultimo commit dando la possibilità di cambiare il commento. Tutto il contenuto della staging area viene inglobato nel commit ripristinato.



# GIT IGNORE

---

GIT mette a disposizione un file particolare, denominato `.gitignore`, che permette alla repository GIT di ignorare file e cartelle descritte in tale file. In questo modo questi file non vengono tracciati.

Per generare facilmente un file `.gitignore`: <https://gitignore.io/>

# CONFIGURAZIONE REMOTA

---

Per fare in modo che la propria repository possa comunicare con un'altra repository remota è necessario configurare un link remoto. Il comando per gestire i link remoti è **git remote**.

- `git remote -v`
- `git remote add "nomeUrlRemoto" <url>`
- `git remote remove "nomeUrlRemoto"`
- `git remote add "nomeUrlRemoto_vecchio" "nomeUrlRemoto_nuovo"`
- NB: il repository "origin" è il repository predefinito remoto quando si crea un progetto GIT clonandolo da una repository remota

# SINCRONIZZARE I CAMBIAMENTI

---

- **git fetch <nomeUrlRemoto>** --> recupera tutta la storia del repository selezionato e NON unisce il mio repository con quello scaricato.
- **git pull <nomeUrlRemoto>** --> recupera tutta la storia del repository selezionato e unisce il mio repository con quello scaricato.
- **git push <nomeUrlRemoto> <branch>** --> carica il contenuto della repository (<branch>) sulla destinazione remota.  
Questa operazione fallisce se la repository non è aggiornata all'ultima versione, in tal caso sarà necessario allineare la propria repository con quella remota.

NB: se non viene specificato nessun nome della repository remota verrà preso quello di default, ovvero "origin". Il branch di default è "master".

# GIT TAG

---

Git permette di contrassegnare con etichette punti specifici della cronologia della versione. Ad esempio nel momento del rilascio di una nuova versione. Queste etichette sono denominate tag e il comando **git tag** permette di gestire queste etichette.

- **git tag "nomeTag"**

Per condividere in remoto un tag si dovrà fare un commit di esso:

- **git commit <nomeBranch> "nomeTag"**
- **git commit <nomeBranch> --tags** (carica tutti i tags locali)

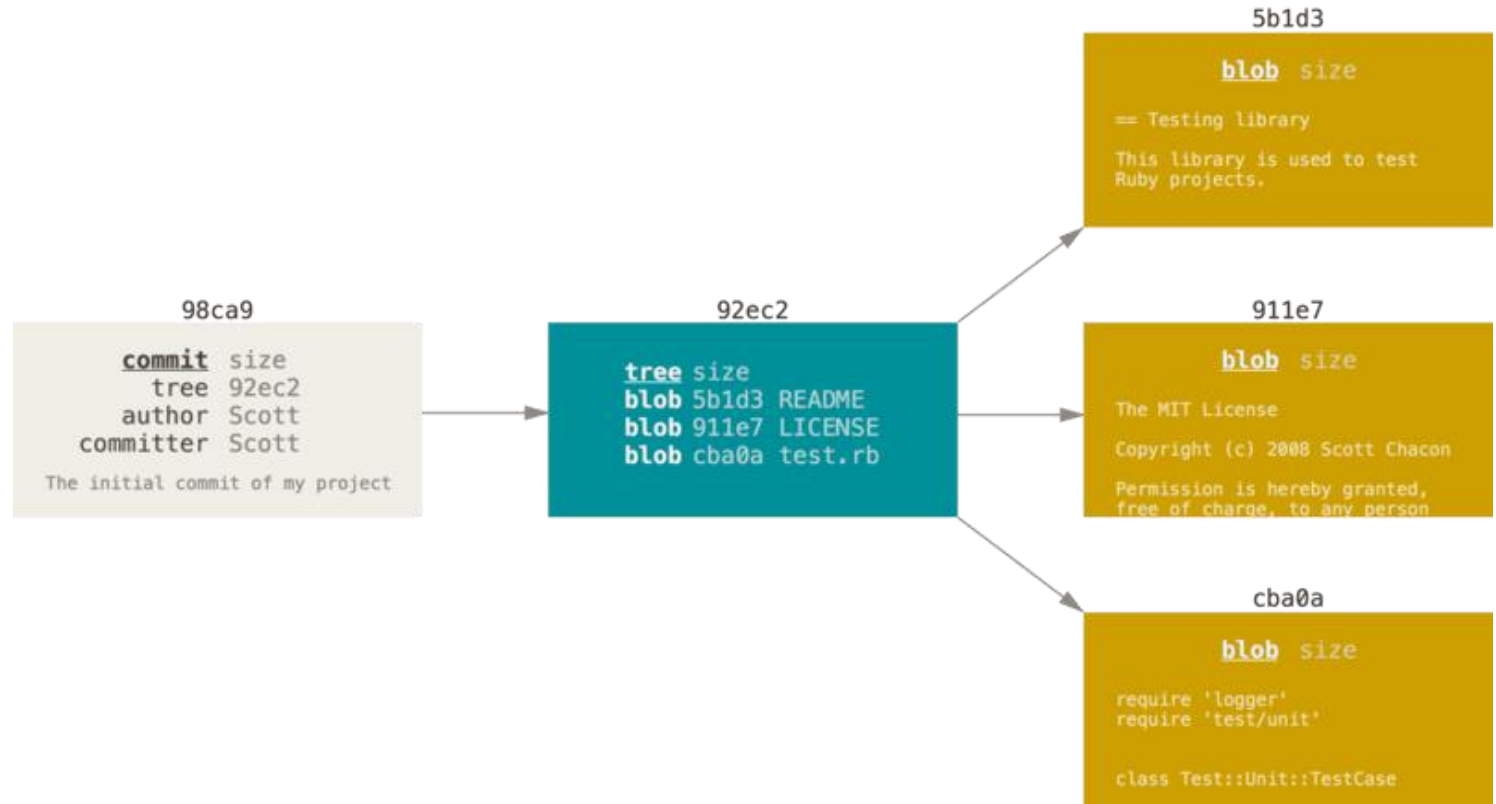
# 3. GIT BRANCHING

---

# STRUTTURA DI UN COMMIT

```
$ git add README test.rb LICENSE
$ git commit -m "The initial
commit of my project"
```

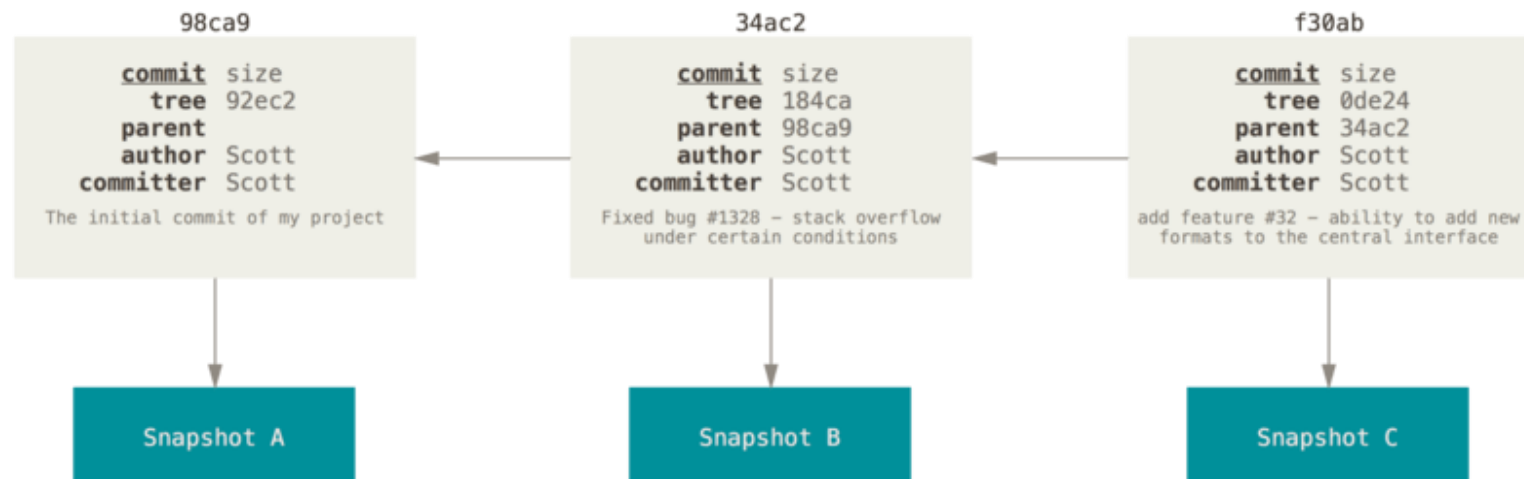
Il commit effettuato con tali comandi genera 5 oggetti nella propria repository git: 3 blobs, ognuno dei quali rappresenta il contenuto dei file modificati, la struttura dati che contiene tali blobs, e i metadati del commit.



# STRUTTURA DI UN COMMIT

Dal secondo commit in poi, l'oggetto che contiene i metadati contiene un nuovo valore che punta al commit precedente.

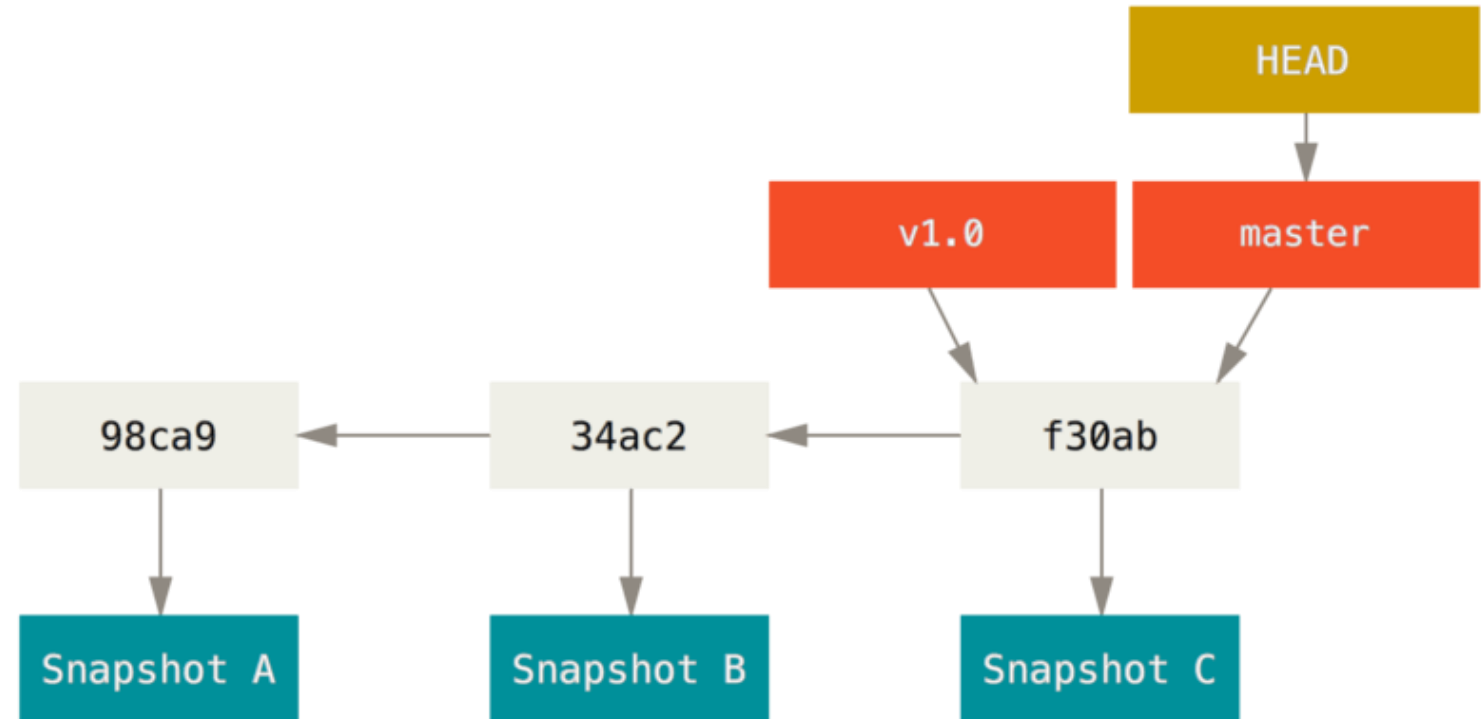
In questo modo la struttura che viene a formarsi è quella di una lista di commit. In particolare questa struttura dati viene chiamata albero e la lista formata dai commit è chiamata ramo, **branch**.



# GIT BRANCH

Il branch di default è chiamato master.

**HEAD** è un puntatore speciale (locale) di git che permette di identificare in quale branch ci troviamo. Punta sempre all'ultimo commit.





# GIT BRANCH

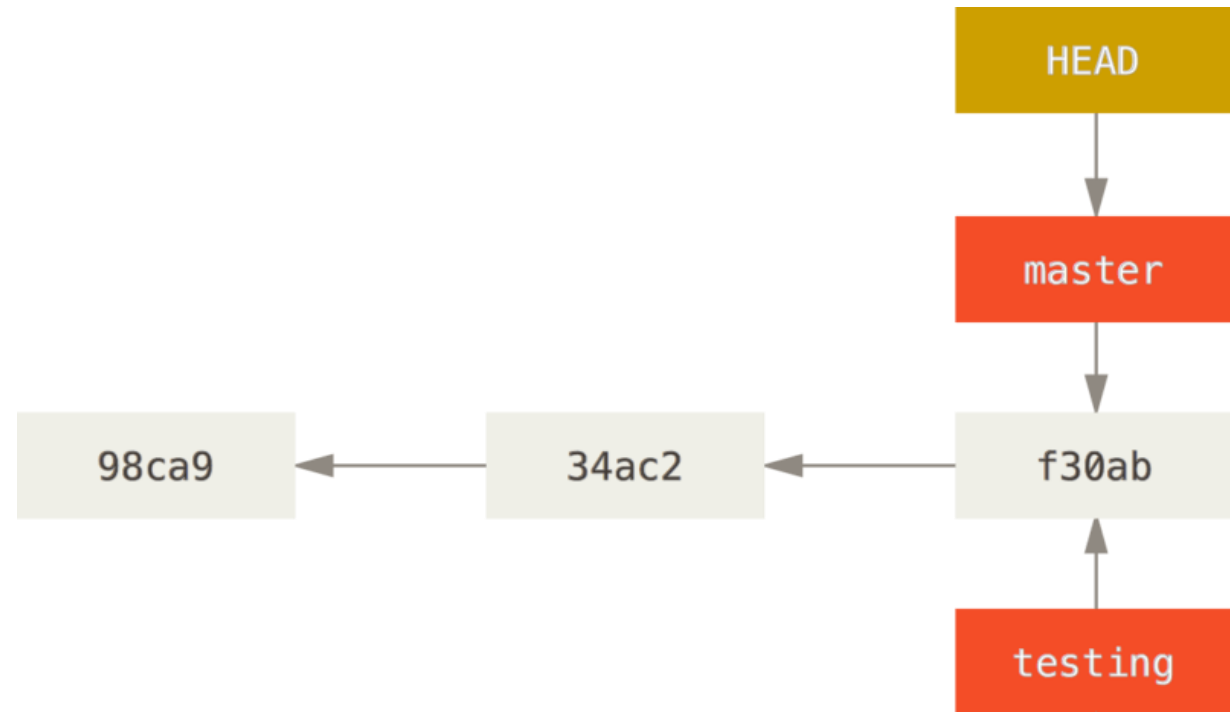
Il comando

- **git branch <nomeBranch>**

crea un nuovo branch partendo dal branch in cui attualmente punta HEAD.

**\$ git branch testing**

NB: creare un nuovo branch non sposta HEAD sul branch appena creato



# GIT BRANCH

Per spostarsi sul nuovo branch  
bisogna utilizzare il comando **git  
checkout**.

**\$ git checkout testing**

Il comando checkout può essere  
utilizzato come zucchero sintattico  
per la sequenza dei comandi git  
branch e git checkout.

**\$ git checkout -b testing**

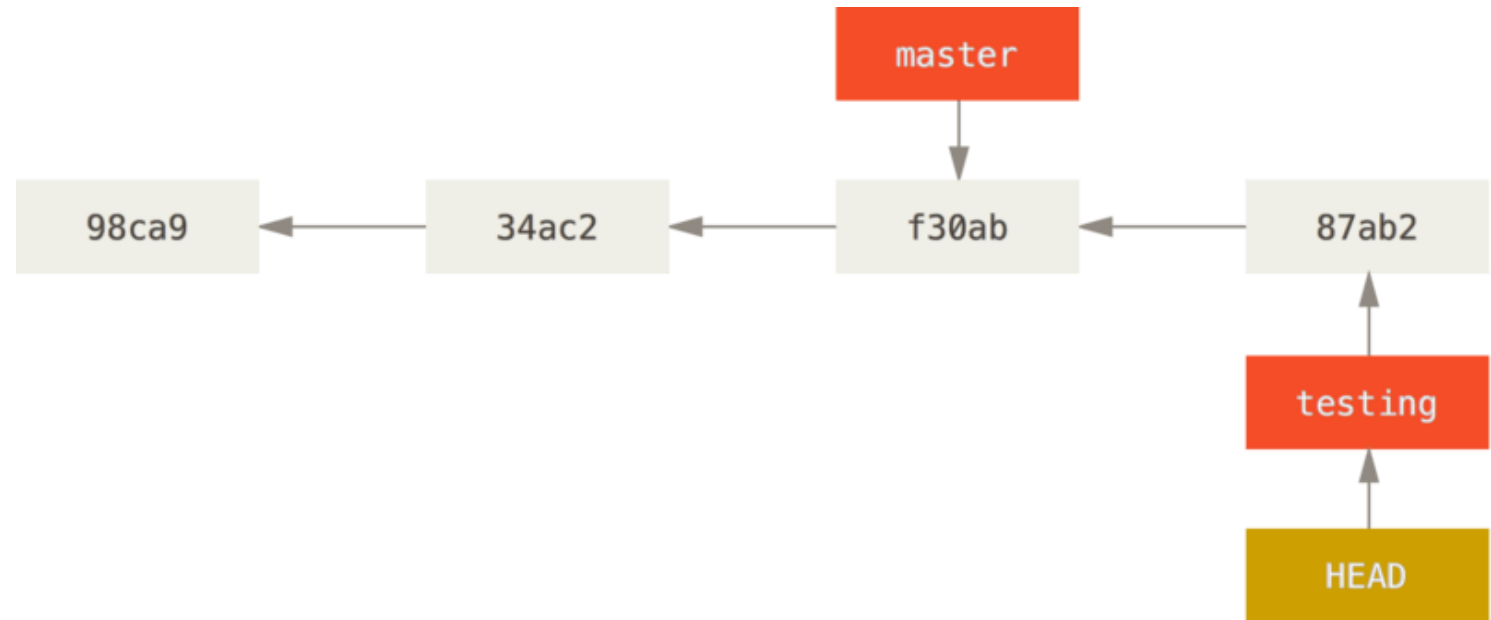


# GIT BRANCH

```
$ vim test.rb
```

```
$ git add test.rb
```

```
$ git commit -m "modifica test.rb"
```



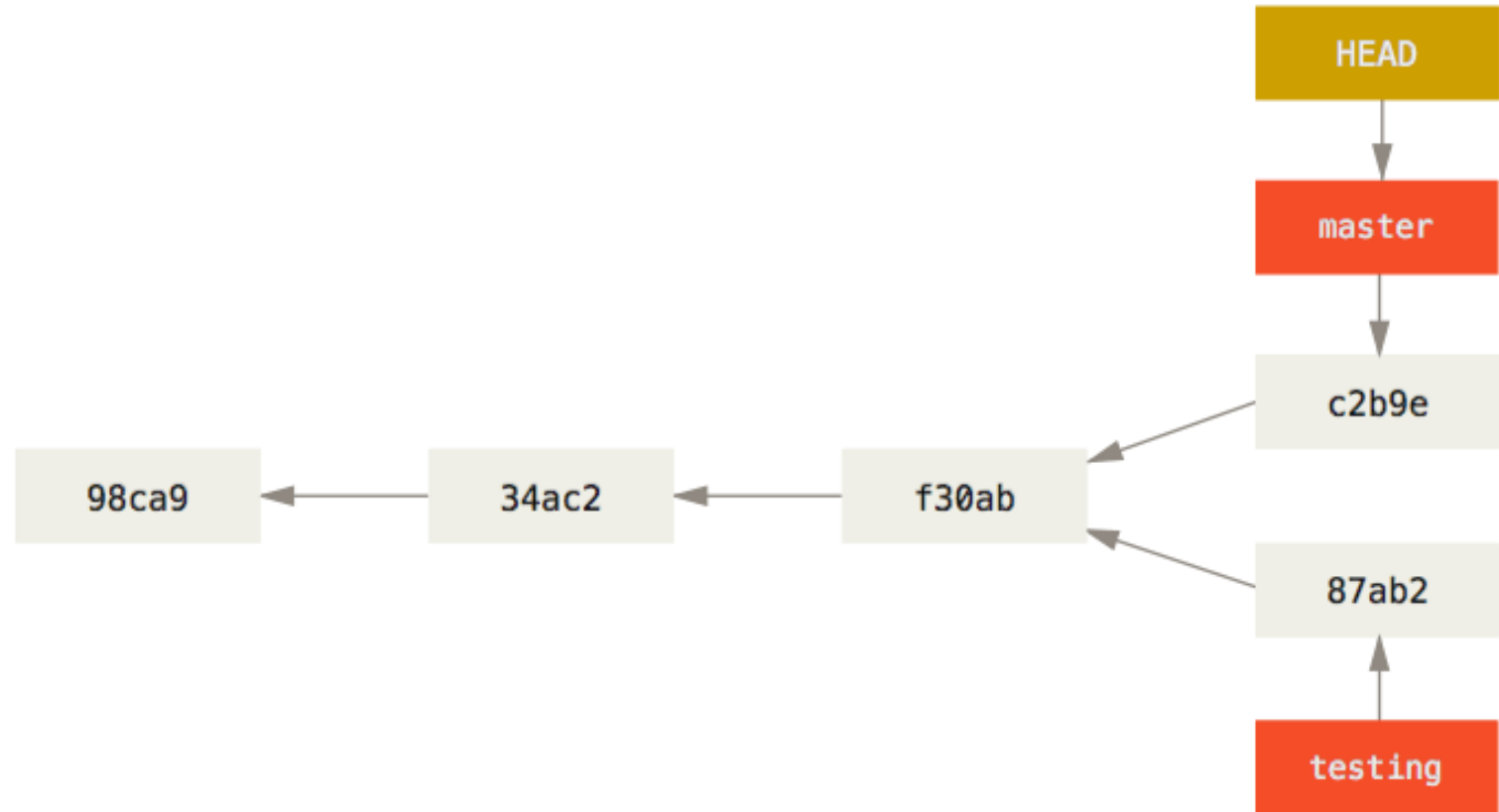
# GIT BRANCH

```
$ git checkout master
```

```
$ vim test.rb
```

```
$ git add test.rb
```

```
$ git commit -m "risolto baco  
all'interno test.rb"
```



# GIT MERGE

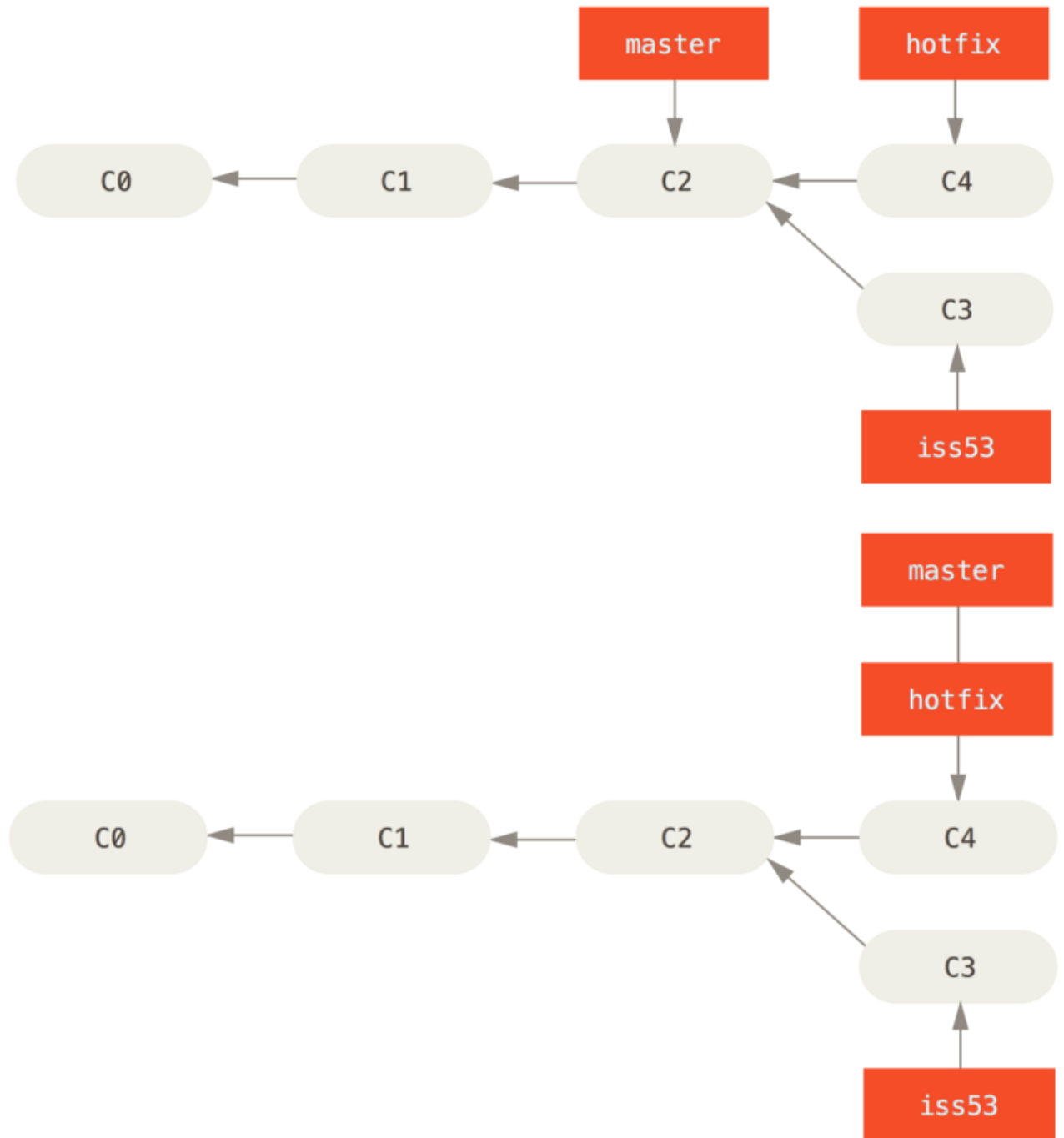
Per unificare due branch si utilizza il comando **git merge <nomeBranch>** che unisce i cambiamenti contenuti nel commit del branch selezionato con il branch in cui è posizionato l'HEAD.

**\$ git checkout master**

**\$ git merge hotfix**

**\$ git branch -d hotfix**

In questo modo il branch hotfix è stato unito al branch master. Questo è un caso particolare perché per eseguire il merge è bastato muovere il puntatore di master avanti di un commit. Questa operazione viene chiamata **fast-forward**

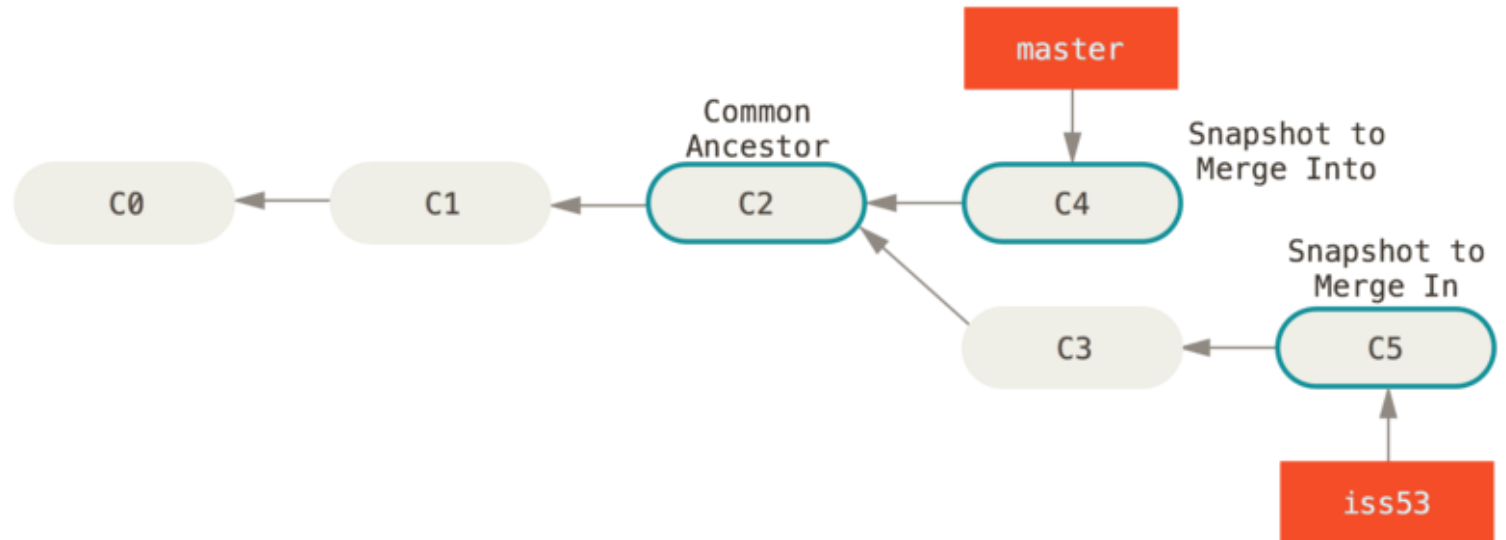
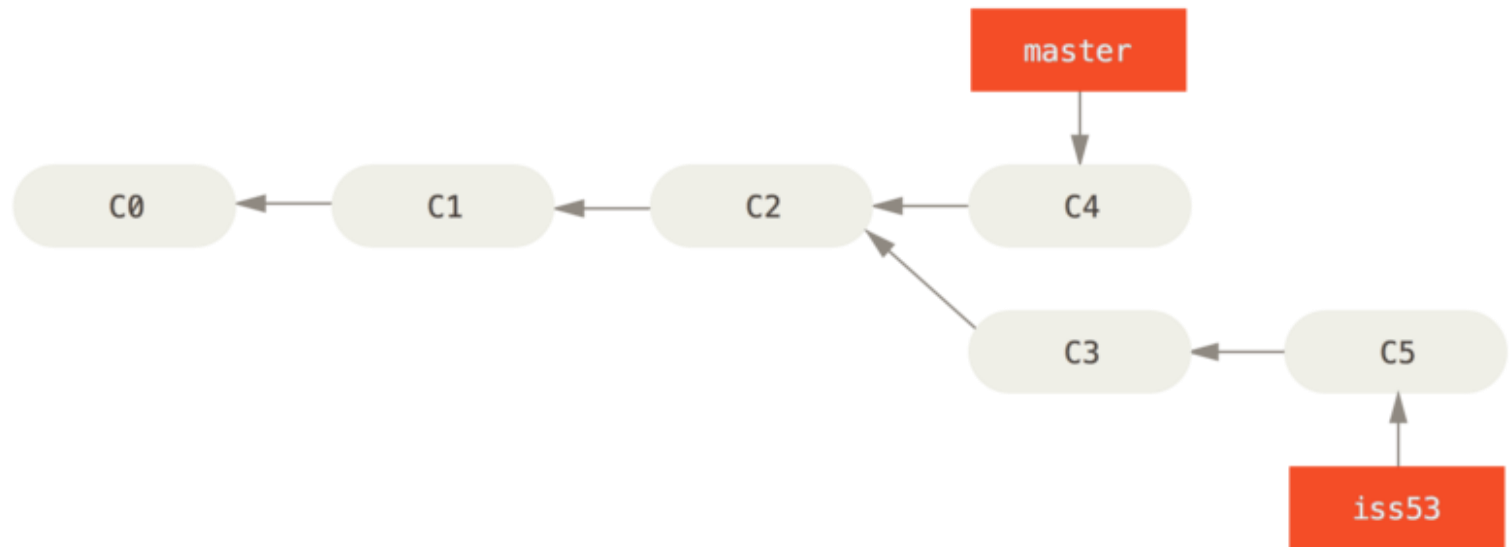


# GIT MERGE

```
$ git checkout master
```

```
$ git merge iss53
```

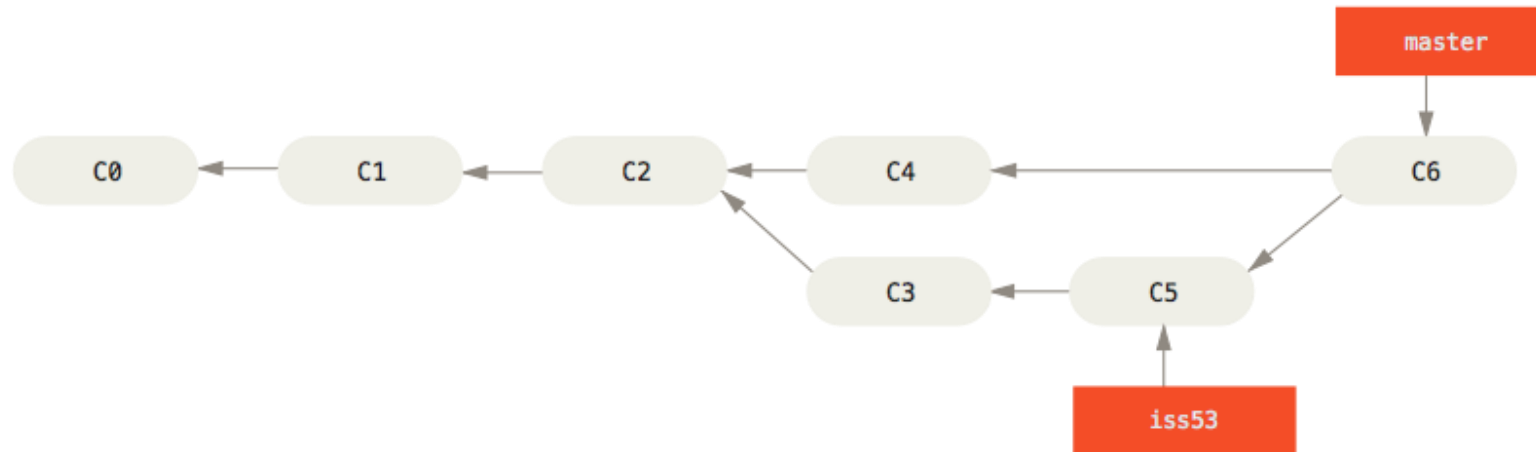
In questo caso git non può eseguire un fast-forward perché la storia di sviluppo si è separata da più di un commit. L'operazione di merge deve creare un nuovo snapshot dei file che è l'unione dei commit HEAD dei due branch e il loro commit antenato in comune.



# GIT MERGE

Git crea un nuovo commit contenente lo snapshot generato dal unione a tre vie dei commit utilizzati per il merge. Questo commit viene chiamato merge commit ed è particolare perché può avere più di un genitore.

Non avendo più bisogno del branch **iss53** lo si può cancellare  
**\$ git branch -d iss53**



# GIT MERGE conflitti

"Occasionalmente" può capitare che un operazione di merge non vada a buon fine. Questo accade quando viene modificato lo stesso file in due branch differenti che stanno per essere uniti; GIT non riesce a capire quali sono le modifiche da mantenere, pertanto è necessario risolvere questi conflitti manualmente

```
davide@t1700: ~/Desktop/work/corso-git
davide@t1700:~/Desktop/work/corso-git$ git checkout -b new_branch
M      link.html
Switched to a new branch 'new_branch'
davide@t1700:~/Desktop/work/corso-git$ vim link.html
davide@t1700:~/Desktop/work/corso-git$ git add link.html
davide@t1700:~/Desktop/work/corso-git$ git commit -m "modifica sperimentale paragrafo
in link.html"
[new_branch 6efc4bb] modifica sperimentale paragrafo in link.html
 1 file changed, 3 insertions(+), 1 deletion(-)
davide@t1700:~/Desktop/work/corso-git$ git status
On branch new_branch
nothing to commit, working directory clean
davide@t1700:~/Desktop/work/corso-git$ git checkout master
Switched to branch 'master'
davide@t1700:~/Desktop/work/corso-git$ vim link.html
davide@t1700:~/Desktop/work/corso-git$ git status
On branch master
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:   link.html

no changes added to commit (use "git add" and/or "git commit -a")
davide@t1700:~/Desktop/work/corso-git$ git add link.html
davide@t1700:~/Desktop/work/corso-git$ git commit -m "aggiunto nuovo paragrafo"
[master 96d19bb] aggiunto nuovo paragrafo
 1 file changed, 1 insertion(+)
davide@t1700:~/Desktop/work/corso-git$ git log --oneline --decorate --graph --all
* 96d19bb (HEAD -> master) aggiunto nuovo paragrafo
| * 6efc4bb (new_branch) modifica sperimentale paragrafo in link.html
|/
* 497293b aggiunto link.html al repository
davide@t1700:~/Desktop/work/corso-git$ git merge new_branch
Auto-merging link.html
CONFLICT (content): Merge conflict in link.html
Automatic merge failed; fix conflicts and then commit the result.
davide@t1700:~/Desktop/work/corso-git$
```



# GIT MERGE conflitti

Git modifica i file in conflitto circoscrivendo l'area del file in conflitto con dei caratteri speciali:

- La parte racchiusa tra `<<<<<<< HEAD` e `=====` identifica le modifiche del branch in cui è posizionato HEAD.
- La parte racchiusa tra `=====` e `>>>>>>>` identifica le modifiche del branch che si sta unendo.

```
davide@t1700: ~/Desktop/work/corso-git
<!DOCTYPE html>
<html>
<body>

<h2>HTML Links</h2>
<<<<<<< HEAD
<p>HTML links are defined with the a tag:</p>
<p>Aggiunto nuovo paragrafo</p>
=====
<p>Questo paragrafo è stato modificato</p>

<h3>Un sottotitolo a caso ma un pelo più preciso di prima</h3>
>>>>>>> new_branch

<a href="https://www.youtube.com/watch?v=dQw4w9WgXcQ">This is a link</a>

</body>
</html>
```

1,1 Top

```
davide@t1700: ~/Desktop/work/corso-git
<!DOCTYPE html>
<html>
<body>

<h2>HTML Links</h2>

<p>Questo paragrafo è stato modificato</p>
<p>Aggiunto nuovo paragrafo</p>

<h3>Un sottotitolo a caso ma un pelo più preciso di prima</h3>
<a href="https://www.youtube.com/watch?v=dQw4w9WgXcQ">This is a link</a>

</body>
</html>

~
~
~
-- INSERT --
```

7,44-43 All

# GIT MERGE conflitti

Una volta risolti tutti i conflitti  
basta aggiungere il/i file  
modificati e proseguire con il  
commit del merge:

**\$ git add <file>**

**\$ git commit**

Il comando commit non vuole  
il flag del messaggio perché  
verrà fornito di default quello  
di merge.

```
davide@t1700: ~/Desktop/work/corso-git
davide@t1700:~/Desktop/work/corso-git$ git merge new_branch
Auto-merging link.html
CONFLICT (content): Merge conflict in link.html
Automatic merge failed; fix conflicts and then commit the result.
davide@t1700:~/Desktop/work/corso-git$ vim link.html
davide@t1700:~/Desktop/work/corso-git$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   link.html

no changes added to commit (use "git add" and/or "git commit -a")
davide@t1700:~/Desktop/work/corso-git$ git add .
davide@t1700:~/Desktop/work/corso-git$ git commit
[master a6dee7a] Merge branch 'new_branch'
davide@t1700:~/Desktop/work/corso-git$ git log --oneline --decorate --graph --all
* a6dee7a (HEAD -> master) Merge branch 'new_branch'
|
| * 6efc4bb (new_branch) modifica sperimentale paragrafo in link.html
| * | 96d19bb aggiunto nuovo paragrafo
|/
* 497293b aggiunto link.html al repository
davide@t1700:~/Desktop/work/corso-git$
```

```
davide@t1700: ~/Desktop/work/corso-git
GNU nano 2.5.3 File: ...avide/Desktop/work/corso-git/.git/COMMIT_EDITMSG

Merge branch 'new_branch'

# Conflicts:
#   link.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master

[ Read 19 lines ]
^G Get Help  ^O Write Out  ^W Where Is  ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit      ^R Read File  ^\ Replace   ^U Uncut Text ^T To Spell   ^_ Go To Line
```

# GIT BRANCH management

Il comando `git branch` fa più che solo creare ed eliminare branch:

**\$ `git branch -v`**

Lista ogni branch con il messaggio e l'hash dell'ultimo commit

**\$ `git branch --merged`**

**\$ `git branch --no-merged`**

Filtra la lista dei branch mostrando rispettivamente solo quelli che sono stati uniti e quelli che non lo sono.

```
davide@t1700: ~/Desktop/work/corso-git
davide@t1700:~/Desktop/work/corso-git$ git branch
* master
  testing
davide@t1700:~/Desktop/work/corso-git$ git branch -v
* master    a6dee7a Merge branch 'new_branch'
  testing    d1a1d64 Modificato paragrafo per il branch testing
davide@t1700:~/Desktop/work/corso-git$ git branch --merged
* master
davide@t1700:~/Desktop/work/corso-git$ git branch --no-merged
  testing
davide@t1700:~/Desktop/work/corso-git$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
davide@t1700:~/Desktop/work/corso-git$ git branch -D testing
Deleted branch testing (was d1a1d64).
davide@t1700:~/Desktop/work/corso-git$
```

# GIT REBASE

Esiste un altro metodo per integrare le modifiche di un branch in un altro:

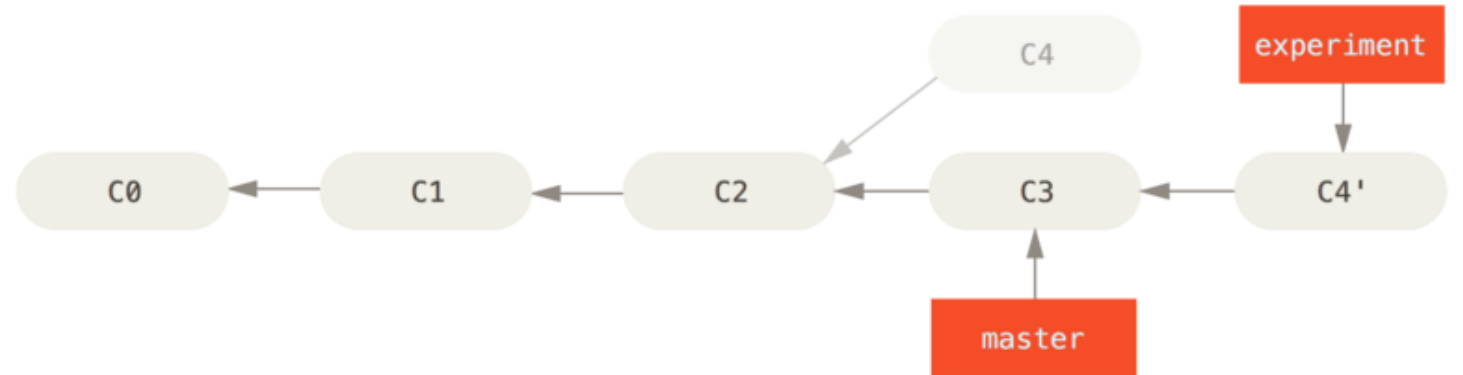
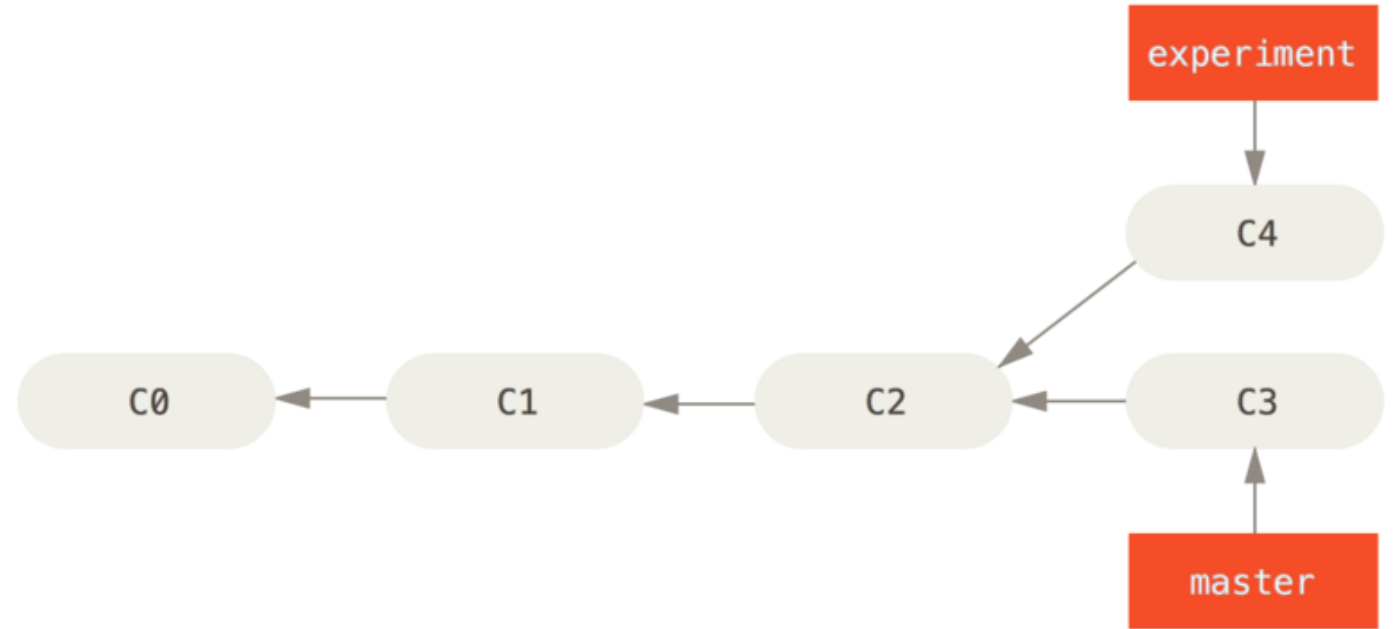
- **git rebase <branch>**

Dal branch che si vuole integrare si esegue il comando e si modifica tutto il branch allineandolo all'ultimo commit del branch target.

**\$ git checkout experiment**

**\$ git rebase master**

NB: il rebase modifica la storia della repository.



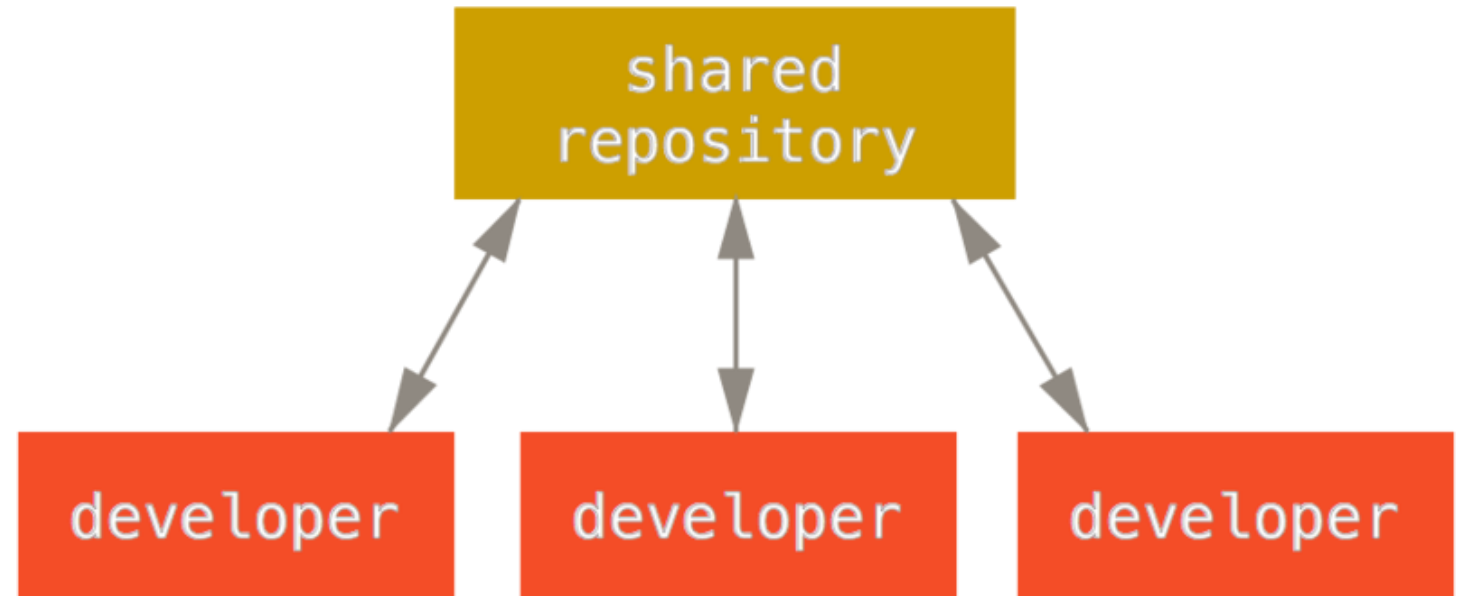
# 4. GIT DISTRIBUITO

---

## CENTRALIZED WORKFLOW

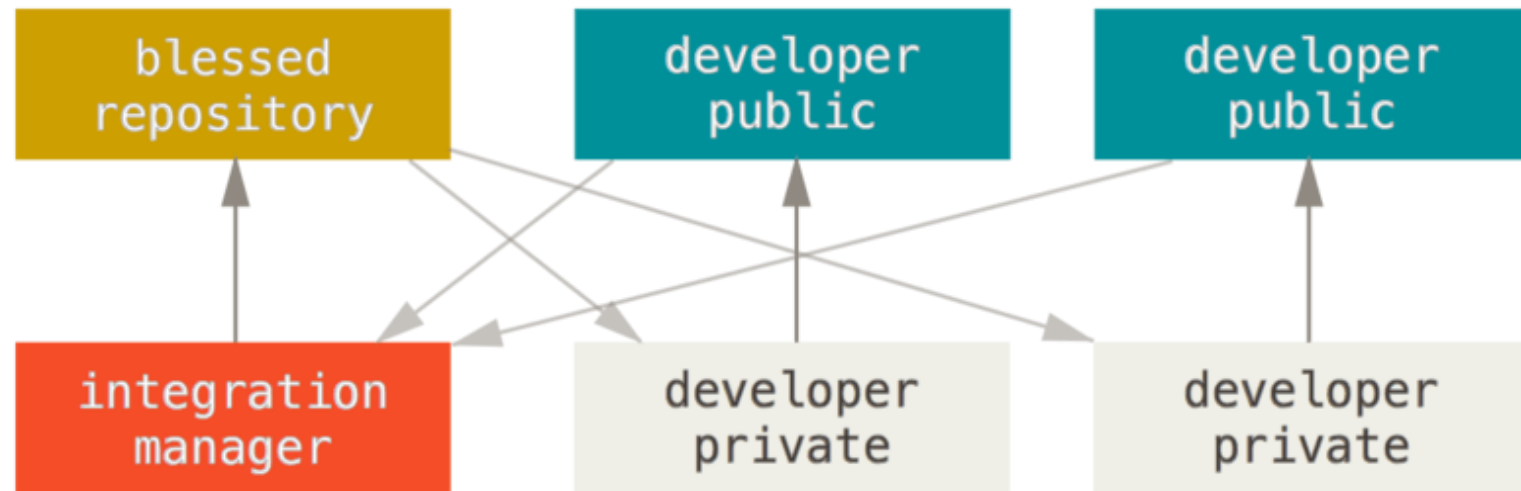
È un modello in cui esiste un solo repository principale con cui gli sviluppatori devono sincronizzarsi.

Questo significa che se due sviluppatori clonano un progetto dalla repository ed entrambi fanno dei cambiamenti, il primo sviluppatore riesce a fare un push dei suoi cambiamenti senza problemi, mentre il secondo deve prima eseguire un fetch, poi un merge e infine può fare un push.



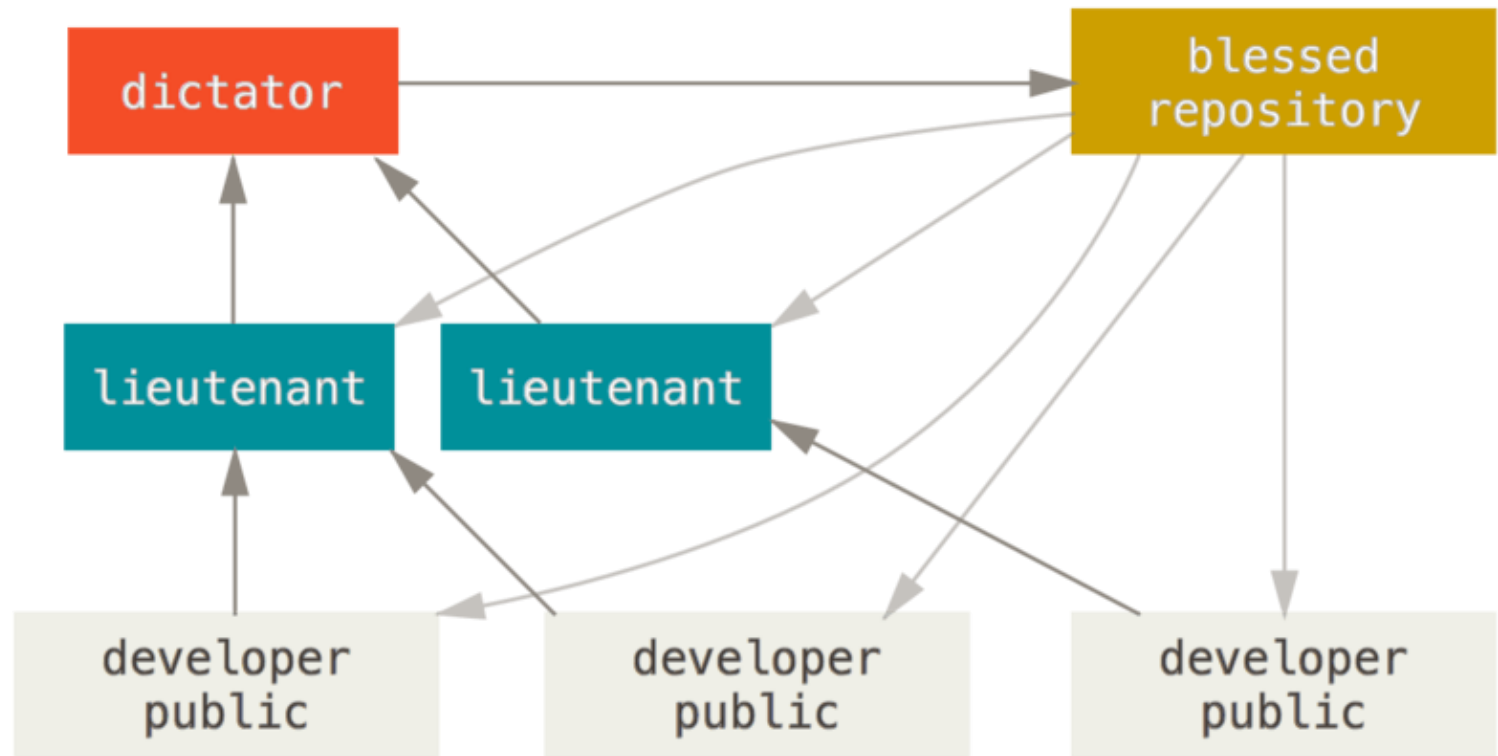
## INTEGRATION-MANAGER WORKFLOW

1. Il gestore del progetto esegue push sulla repository pubblica del progetto.
2. Un contributore clona la repository pubblica e fa dei cambiamenti.
3. Il contributore esegue push sulla propria repository pubblica.
4. Il contributore richiede al gestore del progetto di fare un pull dei cambiamenti.
5. Il gestore aggiunge la repository del contributore come nuovo remote e esegue un merge localmente.
6. Il gestore esegue un push dei cambiamenti sulla repository principale.



## DICTATOR AND LIEUTENANTS WORKFLOW

1. Gli sviluppatori lavorano su uno specifico branch.
2. I luogotenenti eseguono un merge del merge specifico sul loro master branch
3. Il dittatore esegue un merge del master dei luogotenenti all'interno del proprio master locale.
4. Il dittatore esegue un push del master locale sulla repository di riferimento.





# RESOURCES

---

- **Git:** <https://git-scm.com/book/en/v2/>
- **Gitflow:** <https://danielkummer.github.io/git-flow-cheatsheet/>
- **GitHub guides:** <https://guides.github.com/>
- **GitHub 101:** <https://guides.github.com/activities/hello-world/>