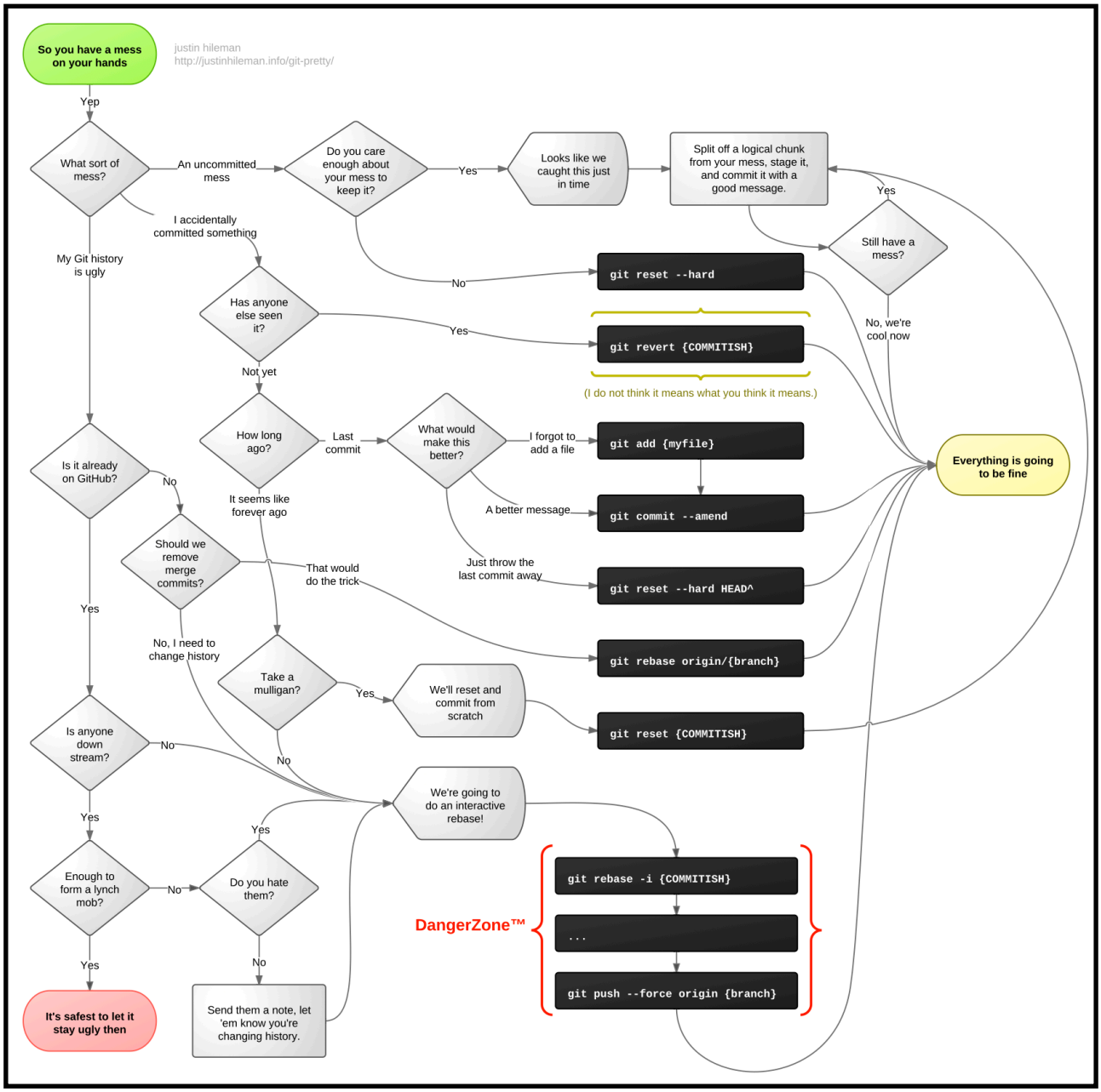
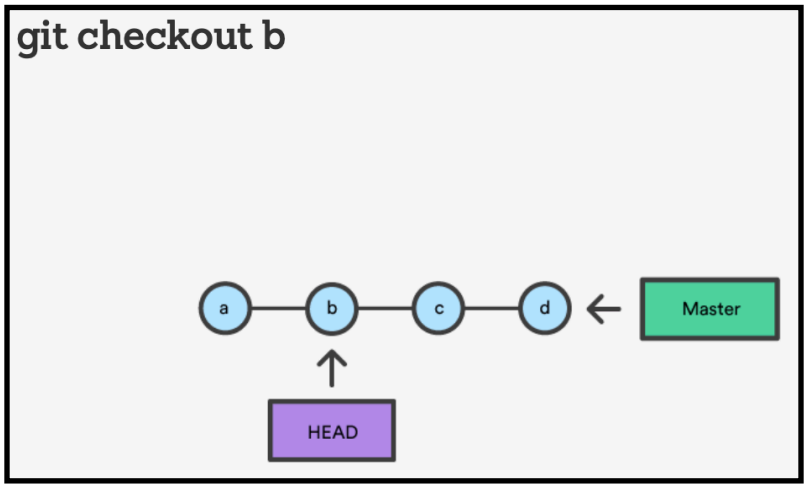
GIT

- **git init**
- **git clone**
- **git diff**
- **git status**
- **git log**
- **git branch**
- **git add**
- **git commit**
- **git merge**
- **git cherry-pick**
- **git remote**
- **git push**
- **git pull**
- **git checkout**



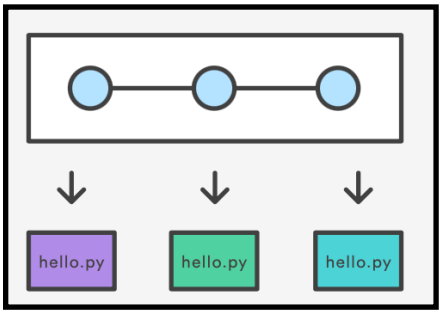
GIT CHECKOUT

- **git checkout <ref> < - - files > :** vengono copiati i <*files*> nella versioni <*ref*> dentro a index e working directory. Se non sono indicati *files* allora viene spostata la HEAD, ma la working directory e index non vengono toccati, sempre che i files nella wd/index non creano conflitti con la configurazione in cui mi voglio spostare. In quel caso bisogna fare commit o stash.



GIT COMMIT

Commits are the core building block units of a Git project timeline. Commits can be thought of as snapshots or milestones along the timeline of a Git project. Commits are created with the `git commit` command to capture the state of a project at that point in time.

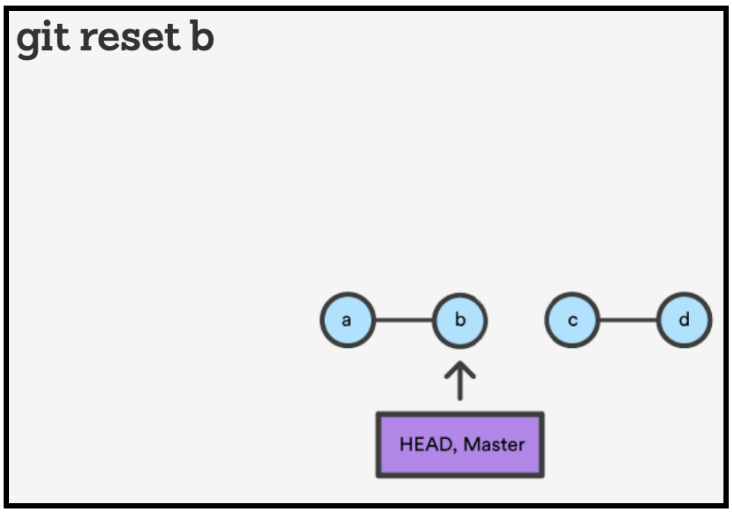


OPTIONS:

- `git commit` : commit the staged snapshot.
- `git commit -a` : commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with `git add` at some point in their history).
- `git commit - -amend` : passing this option will modify the last commit. Instead of creating a new commit, staged changes will be added to the previous commit.

GIT RESET

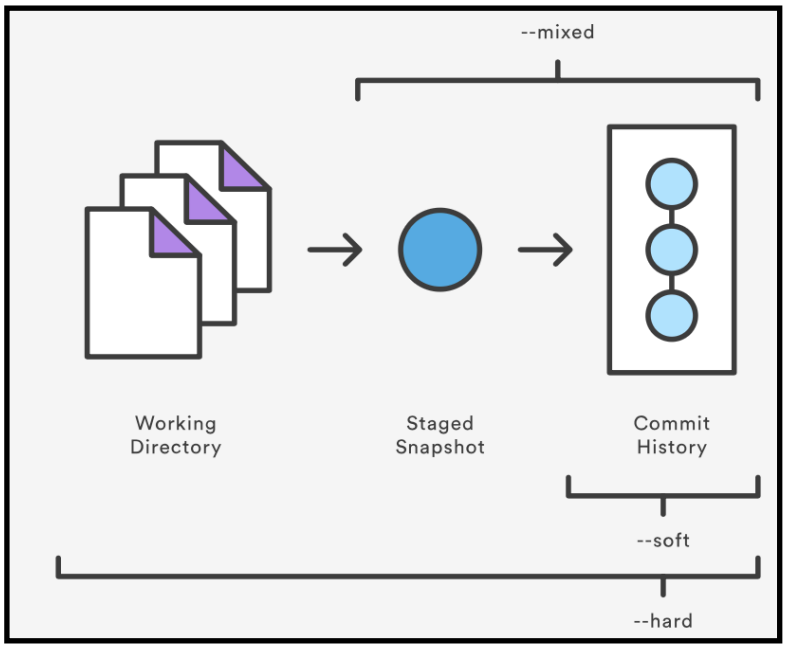
- **git reset <ref> < - - files>**



- **git reset - - mixed**: senza parametri, cambia la HEAD e l'index si svuota, le modifiche fatte da b in avanti vengono salvate nella working directory. È l'analogo all'opzione **- - mixed**, che quindi è quella di default
  file committati -> wd
  file nell'index -> wd
  file nella wd -> wd

- **git reset - - soft HEAD~3 :** sposto il branch a 3 commit prima dell'HEAD. Le modifiche fatte da b in avanti vengono salvate nella wd e nell'index.
  file committati -> index *(i file cambiati nei commit che resetto finiscono nell'index)*
  file nell'index -> index
  file nella wd -> wd

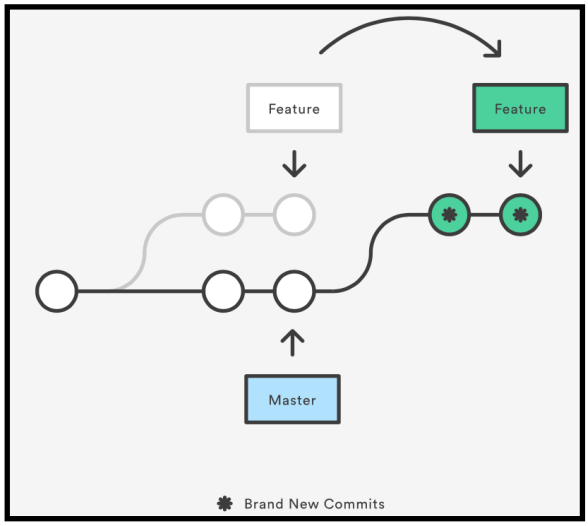
- **git reset - - hard HEAD~3 :** sposto il branch a 3 commit prima dell'HEAD, NON mettendo le modifiche nella wd e index, che quindi si svuotano
  file committati -> 0
  file nell'index -> 0
  file nella wd -> 0 (tranne gli untracked che restano nella wd)



Tutta la confusione fatta negli ultimi commit (3 commit nel nostro esempio) viene messa dentro un unico calderone (wd o wd+index), in ottica di fare ordine ora.

GIT REBASE

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow.

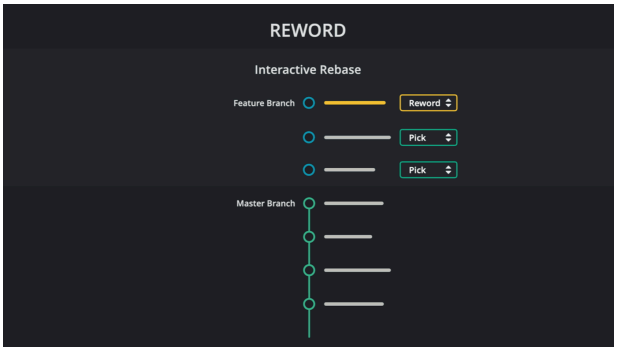


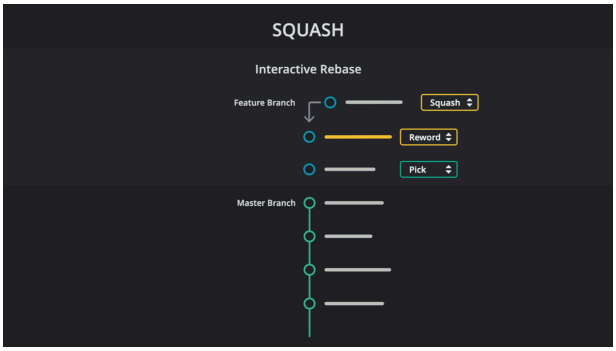
- **git rebase master**
- **git rebase - - root -i**: interattivo della root

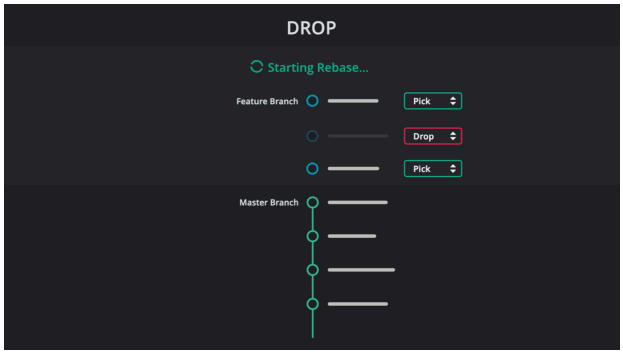
Git Rebase Interactive Tutorial: https://blog.axosoft.com/learning-git-interactive-rebase/

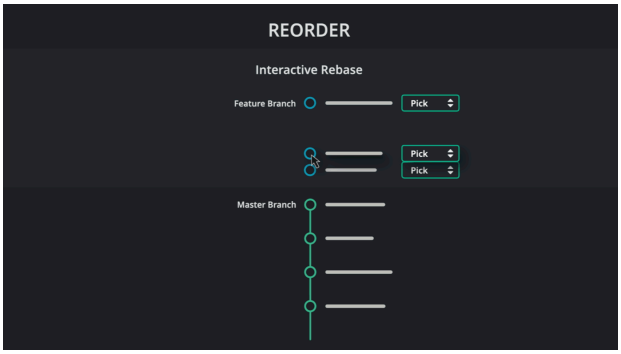
git rebase -i <hash>

Rebase interattivo dal commit <hash> fino alla HEAD. Molto spesso ha senso mettere come argomento HEAD~n al posto di un hash, per navigare nella parentela. La root non viene considerata nell'interattivo, se non da sola, con l'opzione - - root.









GIT BISECT

Utilizzo della ricerca binaria per trovare un bug all'interno della storia dei commit. Utile per trovare un commit che poi modificherò con rebase o altri comandi.

- **git bisect start**
- **git bisect bad**: senza argomento dopo “bad” vuol dire che mi riferisco a HEAD, punto di fine della ricerca
- **git bisect good <hash>**: versione che conosco essere buona, punto di partenza della ricerca
- **git bisect reset**

esempio con 100 commit
99: ultimo commit (HEAD)
...
0: primo commit

git bisect start
git bisect bad
git bisect good 0 [0-99]

=> 50: controlla se è buono o no questo commit, quindi digita git bisect good o git bisect bad di conseguenza

git bisect bad [0-50]

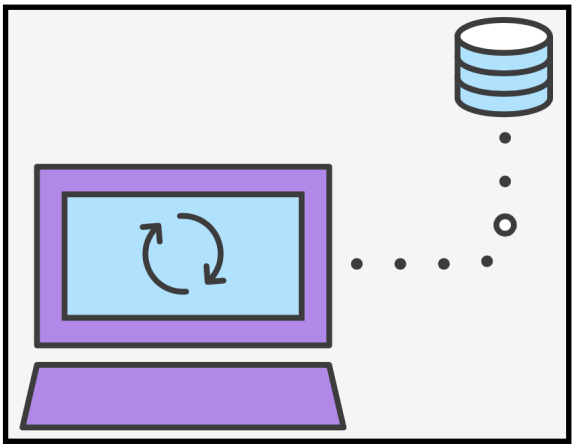
=> 25 ...

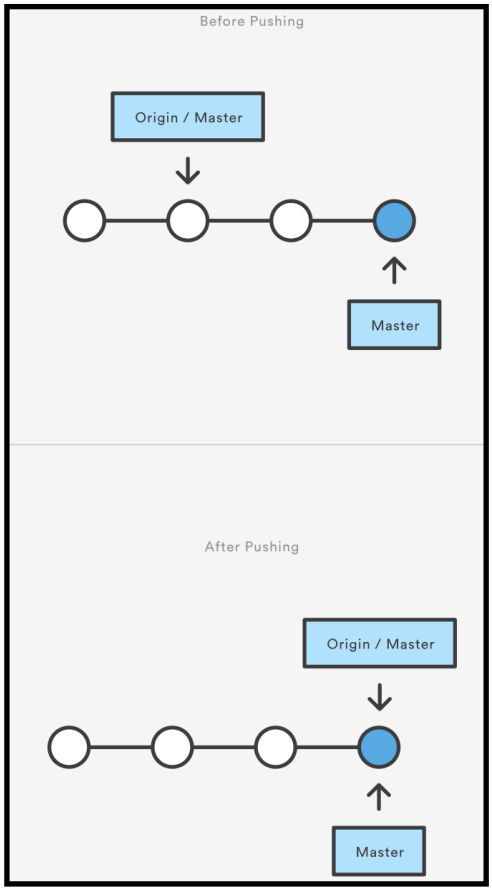
esempio versione con scritto shell
git bisect run sh -c “grep <word> <filename>”

Il comando interpreta il risultato dello script, quindi della grep. Grep ritorna 0 se ha successo, 1 altrimenti. Quindi 0-> **git bisect good**, 1-> **git bisect bad**.

GIT PUSH

- **git push**
- **git push - - force origin <branch>**: forzare la scrittura sul repo centrale (origin). Per limitare i danni, GitHub (come altri gestori) ha introdotto dei concetti di diritti sul repo in base alla propria posizione

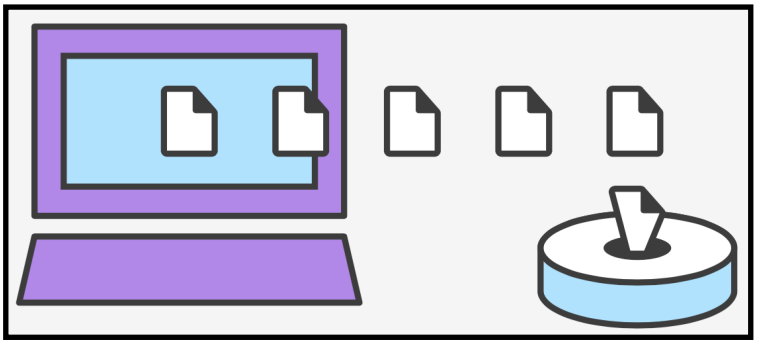




GIT STASH

Mette “da parte” working space e index e poi fa un reset. Ci sono diverse situazioni in cui può tornare utile utilizzare questo comando:

1. Interrupted workflow
2. Pulling into a dirty tree
3. Testing (partial) commits



1. GIT STASH - INTERRUPTED WORKFLOW

Il mio capo mi chiede di fare qualcosa urgentemente, però avevamo qualcosa sulla nostra working directory e index su cui stavamo lavorando. Devo lavorare su una situazione che non è quella su cui sto lavorando ora.

```
git stash
*edit emergency fix*
git commit -a -m “fix in a hurry”
git stash pop (- - index)
```

2. GIT STASH - PULLING INTO A DIRTY TREE

Quando provo a fare pull ma la mia versione non è aggiornata con quella remota.

```
git pull
file foobar not up to date, cannot merge
```

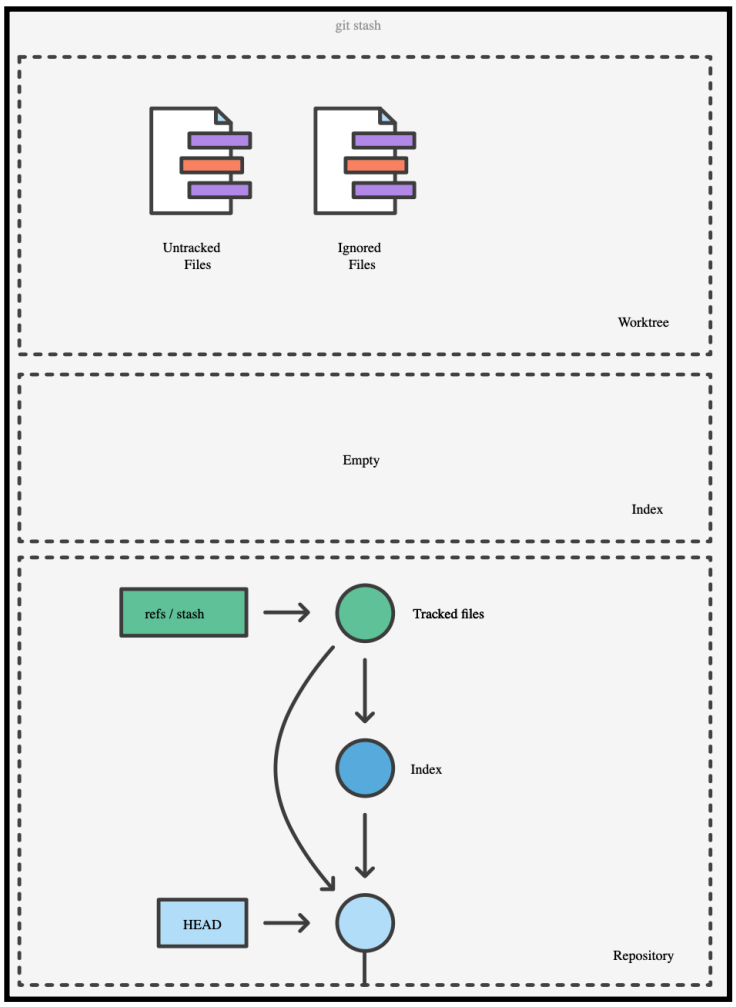
```
git stash
git pull
git stash pop (- - index)
```

3. GIT STATSH - TESTING (PARTIAL) COMMITS

Si vuole testare un index che si vuole committare, diverso dalla working directory, perchè un test viene fatto su quello che c’è nella working directory.

```
git stash - - keep-index
*edit, test ecc.*
git commit -m “tested”
git stash pop
```

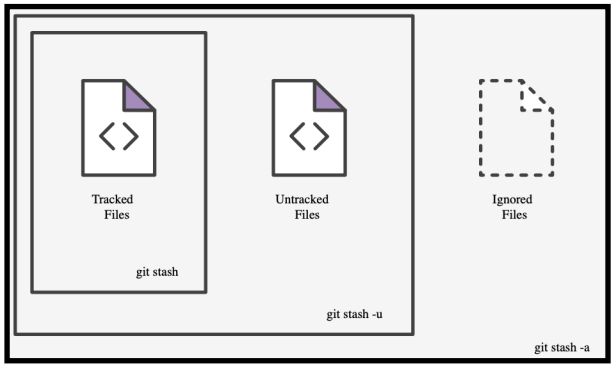
COME FUNZIONA UN GIT STASH?



Vengono creato un branch (**ref/stash**) e due commit su questo branch, uno per l'index e uno per la working dir.
Ovviamente, dopo git stash pop, potrebbero esserci dei conflitti, nel caso in cui io sia andato a toccare dei file di cui ho fatto stash precedentemente. Alla fine git crea un branch con due commit, quindi quando fa git pop è come se eseguisse una sorta di reset e poi si spostasse all'ultimo commit.

In caso di conflitto al momento di **git stash pop**, bisogna risolvere il conflitto (per esempio manualmente) e poi eseguire **git add** e **git commit**. Non dimenticare di eseguire **git stash drop** per eliminare lo stash, perchè questo non viene eliminato automaticamente in caso di merge dopo il pop.

OPTIONS

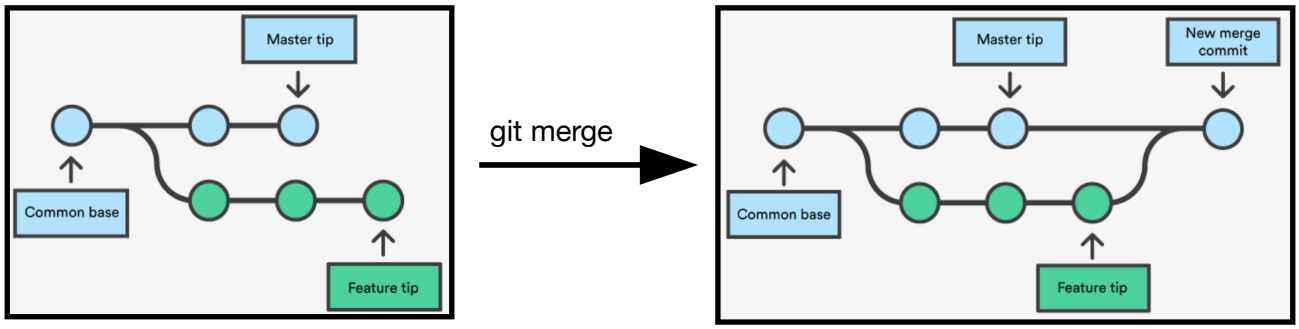


- **git stash -u**
- **git stash -a**
- **git stash pop - -index:** this option is only valid for pop and apply commands. Tries to reinstate not only the working tree's changes, but also the index's ones
- **git stash - -keep-index:** all changes already added to the index are left intact
- **git stash apply:** like pop, but do not remove the state from the stash list

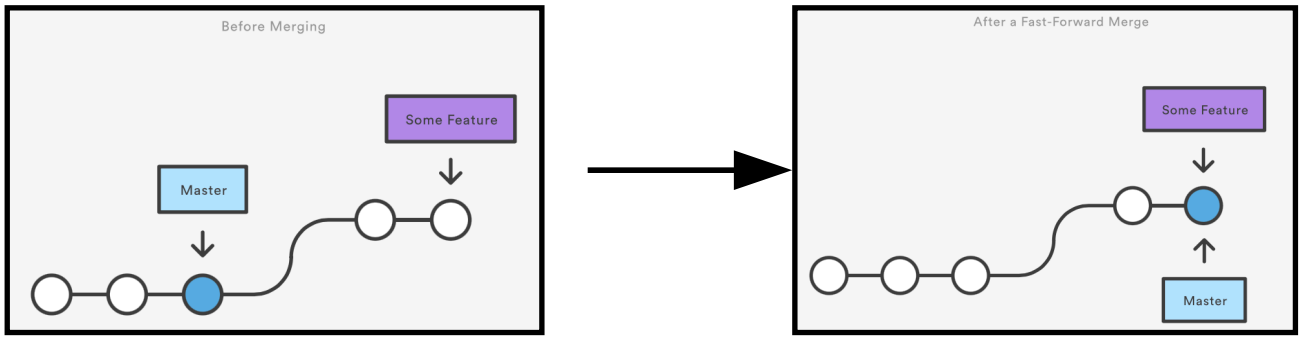
GIT MERGE

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, `git merge` is used to combine two branches. In these scenarios, `git merge` takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.

3-WAY MERGE



FAST FORWARD MERGE

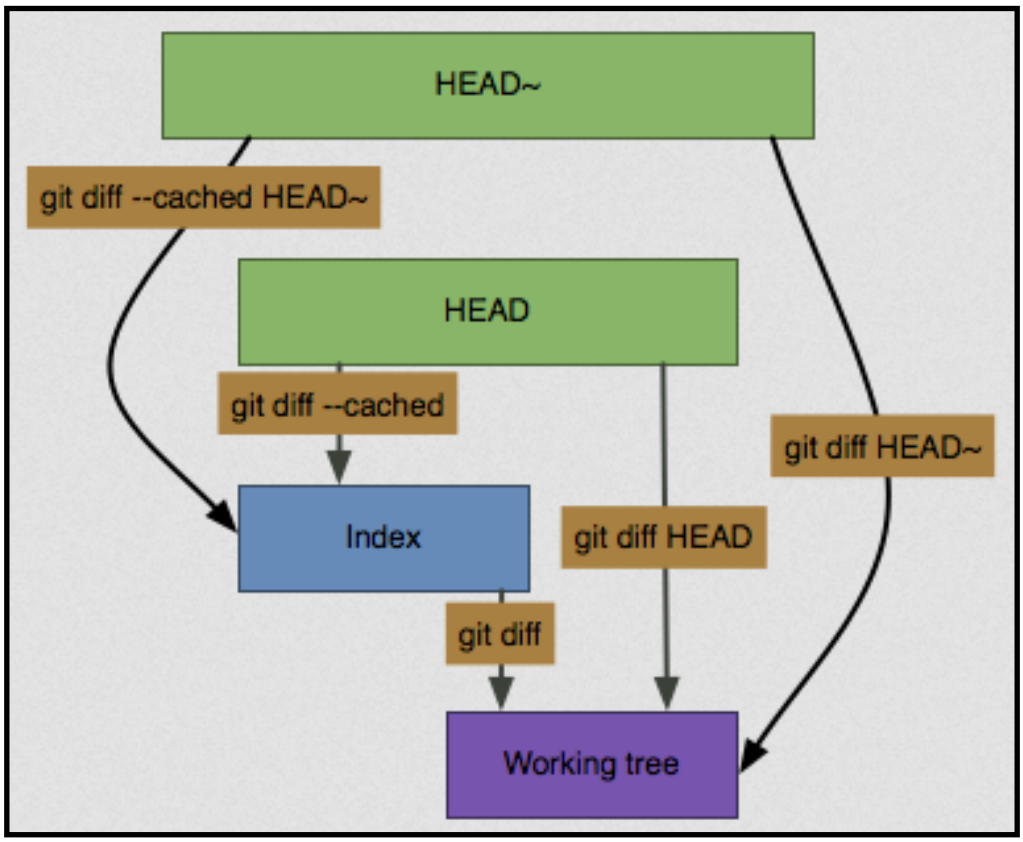



```
git merge --no-ff <branch>
```

This command merges the specified branch into the current branch, but always generates a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

GIT DIFF

- **git diff**: index vs working directory. Differenze tra diverse versioni o tra diversi file di diverse versioni
- **git diff <ref1> <ref2>**
- **git diff - - cached**: (HEAD vs index)
- **git diff - - files**: confronto solo su uno o più files



ALTRO

- **git cat-file -s *<hash>*** (vanno bene anche solo i primi 4 numeri dell'hash) (size)
- **git cat-file -t *<hash>*** (type - ex. commit, tree …)
- **git cat-file -p *<hash>***
  (questi 3 possono essere eseguiti indipendentemente dalla directory in cui ci troviamo, ma "fa tutto git")
- **git ls-files -s :** show information about files in the index and the working tree. The [-s] option show staged contents' mode bits, object name and stage number in the output
- **git cat-file --batch-check --batch-all-objects**
- **git fsck --unreachable --no-reflogs**

- **cat … | pigz -d | od -c** (decomprimere)
- **git hash-object *<filename>*** (preview dell'hash)
- **git hash-object -w *<filename>*** (bypassare index)
- **git fsck** (file system check)
- **git checkout .** (ripristino alla situazione dell'index)
- **git clone *<repo1> <repo2>***
- **git fetch**

- **git hist - - all**

- **git commit - - amend** (aggiorna l'ultimo commit e lo unisce con i file in stage)
- **git commit -a**: add implicito di tutti i file versionati almeno una volta + commit
- **git merge branchname**
- **git revert :** non distrugge o cambia storia, ma crea un commit che inverte gli effetti del commit citato, che potrebbe anche non essere l'ultimo
- **git add**
- **git add -p**: agisco a livello di hunks e non del file intero
- **git reflog**: history dei log locali
- **git request-pull <hash> <url>**: riassunto dei cambiamenti che si genererebbero a fronte di un pull

- **git filter-branch**: comando molto potente e complicato. Permette di riscrivere la storia in maniera ancora più completa.

Permette di fare come se un file non fosse mai stato versionato. Più potente di git rebase in quanto non linearizza la storia, ma mantiene la struttura potenzialmente complessa e diramata. Praticamente percorre ogni commit, toglie il file se presente e poi ricalcola gli hash.

```
git filter-branch - - index-filter 'git rm - - cached - -ignore-unmatch -<filename>' -a
```

Permette di ricavare una repository con solo il contenuto di una subdirectory

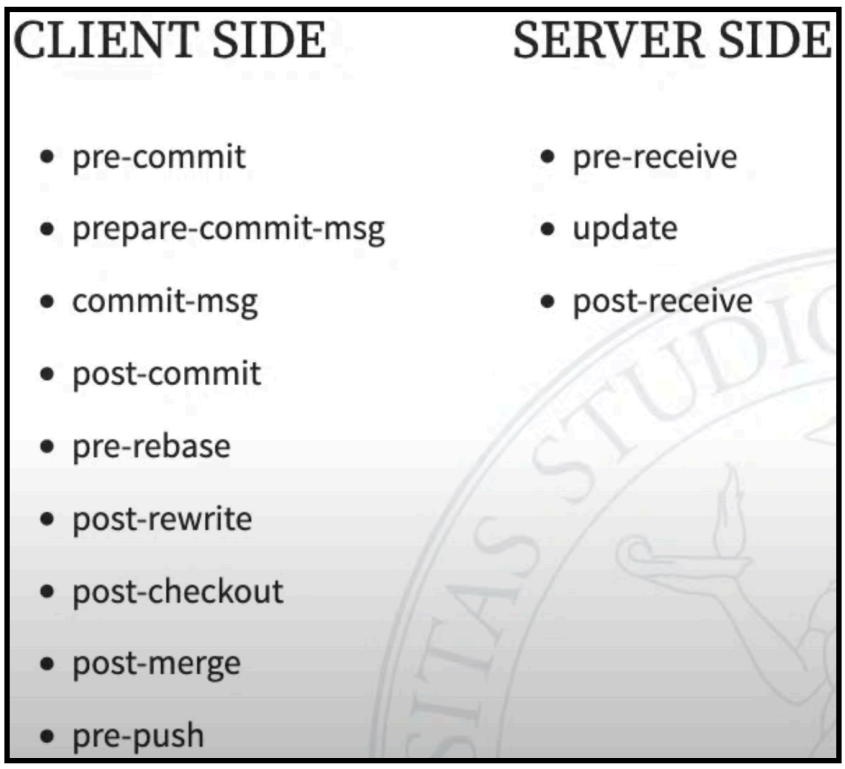
```
git filter-branch - -subdirectory-filter <foodir> - -   - -all
```

Permette di fare come se alcuni file fossero sempre stati in una subdirectory

```
git filter-branch -f - -tree-filter 'mkdir ._p ; mv * ._p; mv ._p core;' - -    - -all
```

HOOKS

Git hooks are scripts that Git executes before or after events such as: commit, push, and receive. Git hooks are a built-in feature - no need to download anything. Git hooks are run locally.



CLIENT SIDE

- **pre-commit**: Viene lanciato subito dopo un commit, prima di editare il messaggio. Quello di default controllo se ci sono errori di spelling nel messaggio di commit.
- **post-commit**

SERVER SIDE

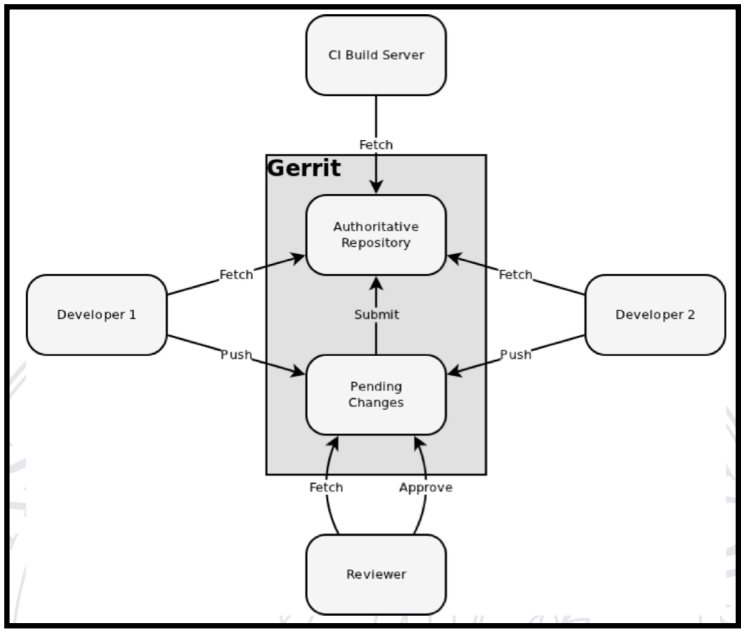
- pre-receive
- update: possono essere usati per rifiutare dei push, differiscono nel nome

- post receive: può essere usato per notifiche

FORK & PULL REQUEST

Un repository pubblico può essere “forkato”, andando quindi a creare altri repository su cui le persone possono lavorarci in privato.
Se queste persone volessero in futuro richiedere di unificare il loro lavoro al lavoro originale possono effettuare delle pull request, ossia delle richieste di merge delle due versioni. Il proprietario originale del progetto potrà accettare o meno la richiesta.

GERRIT

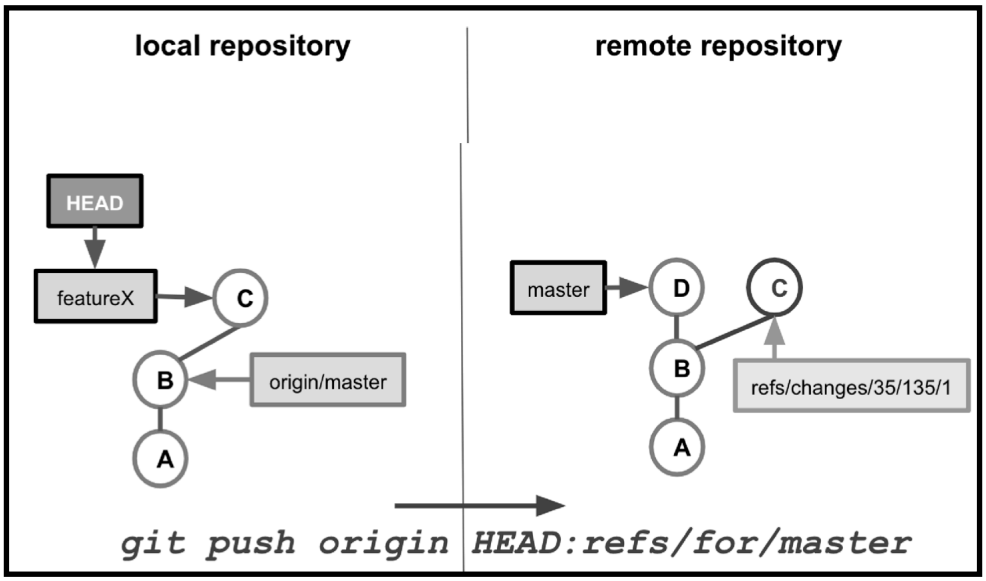


Progetto Google, permette di rendere distribuite le sottomissioni ed il processo di approvazione.

Ogni push sul repository comporta la creazione di un nuovo branch, in attesa di approvazione.

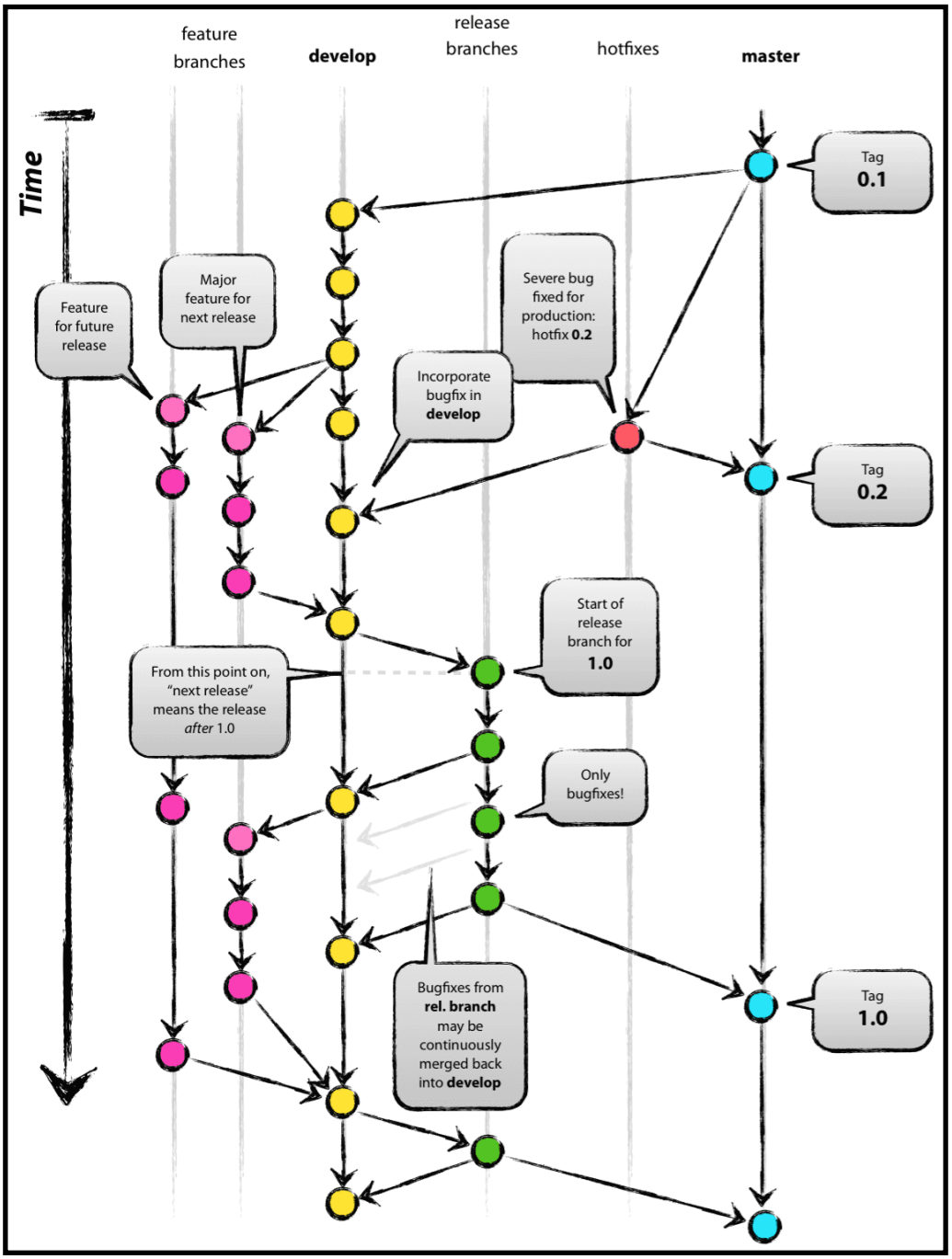
Approver: ruolo che deve decidere se una feature è adatta per il progetto, se questa può causare problemi di integrazioni o problemi di sicurezza, ecc.

Verifier: ruolo che deve fare la build e testare una nuova integrazione

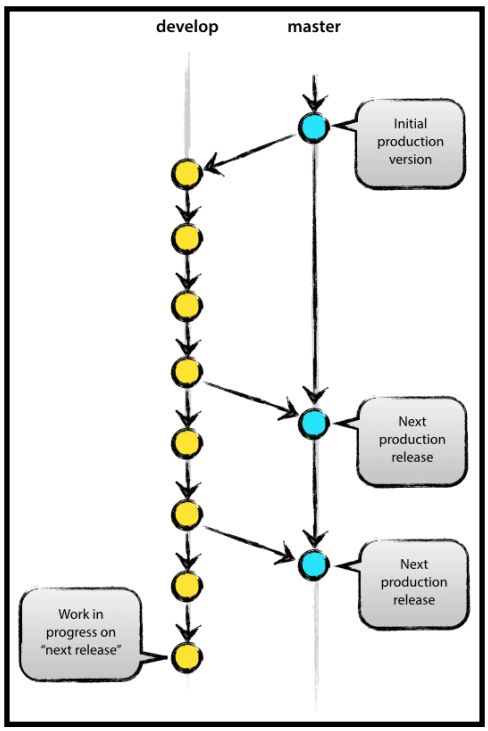


GIT FLOW

GitFlow è un workflow di git che ha come obbiettivo quello di definire uno standard per l'utilizzo dei branch. GitFlow differenzia diversi tipi di branch e definisce nuove operazioni specifiche.



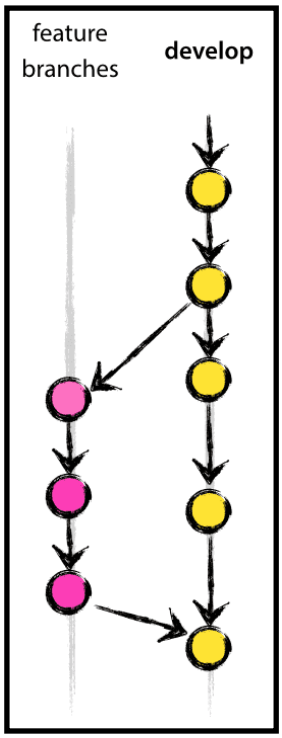
MAIN BRANCHES



Master e **Develop**, due rami con “vita infinita”.

- **Master** deve contiene le versioni stabili e pronte per la consegna. Contiene l’ultima versione stabile per il cliente.
- **Develop** non è il ramo in cui sto sviluppando, ma è il ramo di integrazione. Contiene l’ultima versione stabile per il team di sviluppo.

FEATURE BRANCHES



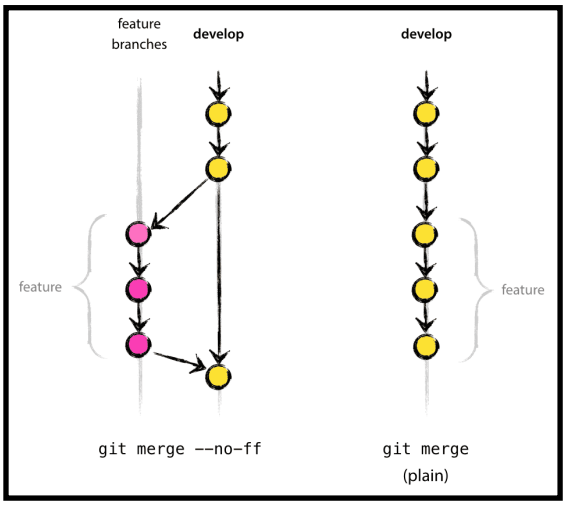
I feature branches (topic branches) sono utilizzati per lo sviluppo di nuove feature, in ottica di una release futura, distante o vicina che sia.

git flow feature start <branchname>
git flow feature finish <branchname>

Le feature finite possono essere unite nel ramo develop per aggiungerle definitivamente alla prossima versione. Il comando di finish corrisponde a:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature
(was 05e9557).
$ git push origin develop
```

Il flag **--no-ff** fa sì che l'unione crei sempre un nuovo oggetto commit, anche se l'unione potrebbe essere eseguita con un fast forward (cioè nel caso in cui il ramo develop fosse un antenato del ramo myfeature). Ciò evita la perdita di informazioni sull'esistenza storica di un ramo di feature e raggruppa tutti i commit che insieme hanno aggiunto la funzionalità.



RELEASE

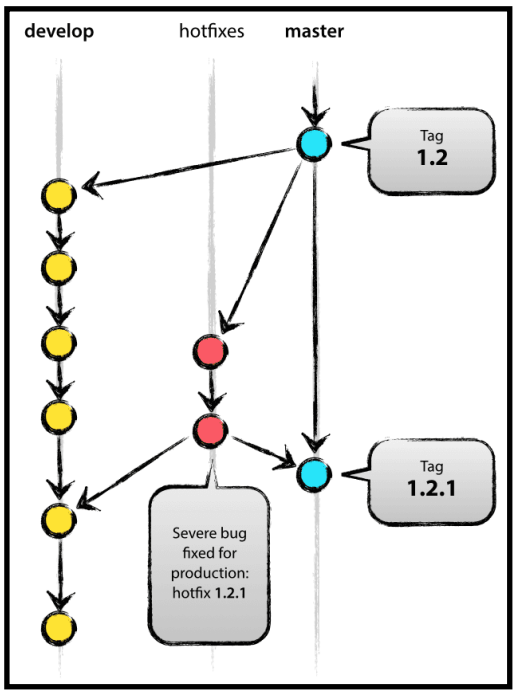
git flow release start <ver>

creazione di un commit sul ramo master. Si congela il lavoro che ora c'è su develop, creando una versione di consegna. Questo branch potrebbe esistere lì per un po', fino a quando il rilascio potrebbe essere implementato definitivamente. Durante questo periodo, le correzioni di bug possono essere applicate in questo ramo (piuttosto che nel ramo develop). L'aggiunta di nuove funzionalità di grandi dimensioni qui è però severamente vietata. Devono essere fusi nello sviluppo e, quindi, attendere la prossima release.

git flow release finish <ver>

Si chiude la release che è stata creata. Viene eseguito un doppio merge (merge su master e back-merge su develop) e l'aggiunta di un tag con commento.

HOTFIXES



Riparazione veloce di bug urgenti, senza aspettare la prossima release.

git flow hotfix start <ver>

È analogo ad un git checkout -b dal master

git flow hotfix finish <ver>

Crea un commit di merge sul master, unificando il master al branch hotfixes. Successivamente riallinea il develop con la nuova versione "senza bug" che c'è sul ramo hotfixes. Attenzione: riallinea develop con hotfixes e non develop con master. Facendo così perdo tutti i test che faccio su master prima di chiudere la release. Develop si riallineerà con master solo quando la release verrà chiusa.