



Contents

Version Control	1
Basic commands	1
local→remote	2
remote→local	2
Separation	2
Moving upwards and laterally	4
Merge conflicts	5

Version Control

It allows to track and navigate development's

- history
- different paths

The project/folder to which you apply it gets called a **repository**.

From here, whenever the user feels like, he can mark selected changes as a **commit**, a “snapshot” of the folder in the form of the Δ with its previous (git-)recorded version.

Git will associate it with a **hash** code, which, in turn, will allow to *go back* to it at any future moment in time.

It's also possible to make the **repository** *remote*, so that it can be interacted with from elsewhere, via its url.

This will allow to work on the same project from different locations, machines, and *people*.

Finally, it allows to *split* the work on the **repo**'s files into separate **branches**, so that it will be possible to work on separate tasks independently and merge later as needed.

Branches, actually, are just pointers to **commits**, hence the good practice “*branch early, and branch often*” is performant, other than easy to apply.

Basic commands

local→remote

- initialize a new **local** repo: `git init`
- create a **remote** repo: go to website
- connect local and remote: `git remote add origin remote_repo_url`
from within the local repo (from now on **origin** stands for the **remote** repo url)
- share your local changes to the remote:
 - `git add file1 file2`
 - aggiungi tutti i file del repo **locale** che sono stati modificati:
`git add -A`
 - `git commit -m "desc"`
 - `git push origin destination.branch_name`
- see the status of your **local** repo with respect to the **remote**: `git status`

```
On branch master
Your branch is up to date with 'origin/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:   .dockerignore
modified:   app/sendGrid/webHookRequest.js

no changes added to commit (use "git add" and/or "git commit
-a")
```

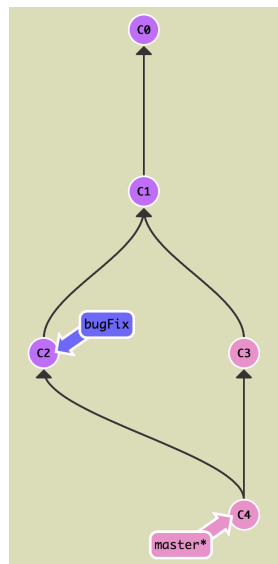
→ useful to see where you are, when you start working on a project with multiple collaborators

remote→local

- create a **local** repo with a copy of the contents of a **remote** repo:
`git clone remote_repo_url`
! → can be used to clone a local repo too
- retrieve **remote** changes to your **local** folder:
`git pull remote_repo branch_name`

Separation

- see the current branches: `git branch -a`
- create a new **local** branch: `git branch branch_name`
- move to (last commit of) another branch: `git checkout branch_name`
- create new **local** branch and move to it: `git checkout -b branch_name`
- push a **local** branch to the **remote**: `git push origin branch_name`
- merge `branch2` into `branch1`: `git merge branch2 from branch1`
- !
 - `git merge master` → merges **local** master into the current branch
 - `git merge origin master` → merges the **remote** master into the current branch



This image is taken from [this nice tutorial](#)

```
git checkout -b bugFix
git commit
git checkout master
git commit
git merge bugFix
```

- delete a branch

locally : `git branch -d localBranchName`

→ `-D` forces the operation

remotely : `git push origin --delete remoteBranchName`

- get **remote** branches on **local**:

```
git fetch
```

```
git fetch -p #when branches were deleted, to retrieve
             the update (p = prune)
```

```
git checkout remote_branch_name #without this the branch
                                 will stay remote
```

Moving upwards and laterally

- see commit hash code from `git log`
! → to go out from the log press `q`
- go to a previous commit: `git checkout 0d1d7fc32`
- create new branch from there:
`git checkout -b branch_name 0d1d7fc32`
- lose **local** changes (e.g. in order to retrieve the new **remote** ones: `git stash` (*riporre*))
- the **HEAD** is the last commit you checked-out to, the one you're working on
 - detach a commit: `git checkout 0d1d7fc32` from there
→ previously the **HEAD** was `HEAD→branch_x→commit_q`, now it is `HEAD→commit_q`
! → making changes from here (and committing them) overwrites them on the **detached** commit not on the last one that was logged for the current **branch**, so this is can be used to **undo operations** (one must make further modifications to be able to commit from there)
 - **git** is smart about hashes: it only requires to specify enough characters of the **hash** until it uniquely identifies the **commit** (e.g. `fed2` instead of the `fed2da64c0efc5293610bdd892f82a58e8cbc5d8`)

- it’s also possible to use **relative operators** to reference a **commit**:
 - * moving upwards of one **commit**: `^` (*caret*)
 - go to parent **commit**: `git checkout branch_name^`
 - grandparent: `branch_name^^`
 - `git checkout HEAD^`
 - * moving upwards of **num** commits: `~num`
 - move (by force) the **branch** 3 **commits** before: `git branch -f branch_name HEAD~3`
- add to the current **branch** the selected commits: `git cherry-pick commit-x commit-y etc`
 ! → e.g.: when debugging many **commits** can be made, but we only want the final one with the fix to be brought into **master**:

```
git checkout debug
/*work*/
git checkout master
git cherry-pick last-commit
```

Merge conflicts

- **vscode** can provide some visual help to solve merge conflicts in general
- but it won’t work with Jupyter notebooks because it has difficulties in rendering it and showing the conflicts at the same time
- the merge operation will show the conflicts in the related files by including both alternatives, the user solves the conflicts by deciding which to delete and which to keep