A very informal introduction to Git

Federico Galatolo

What is not git?

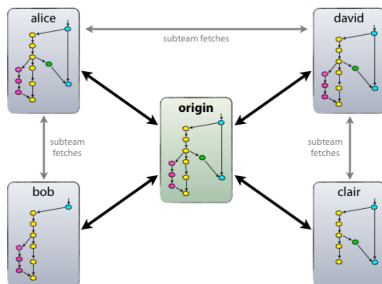
- Dropbox
- Google Drive
- iCloud
- ...

What is git?

Git is VCS (Version Control System).

VCS are about the **changes** not the **files**

Git is more than a VCS



Git is a **distributed** VCS.

Each participant can have a different view of the state of the project.

Repository

A Repository (or “repo”) is a data structure containing all project’s files and history.

You can clone a remote repo with the `clone` command:

```
git clone <repo url>
```

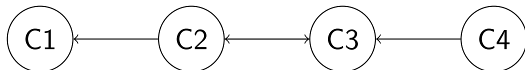
Special files

In a repo you can notice some files starting with `.git`

Those are special files and folders used to store the project history and to instruct git.

.git	A folder containing the project history (do not touch!)
.gitignore	A file containing ignoring rules
.gitmodules	A file containing submodules information

Commit



A **commit** is a **snapshot** of the project in a given time.

Commits are **immutable** and represent a transition from a state to another.

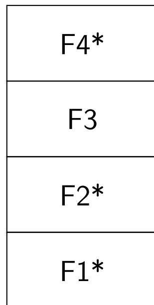
The commits are atomic units of modification within a project.

A commit is uniquely identified by its **hash**

How to commit

A git commit is a two-stages process.

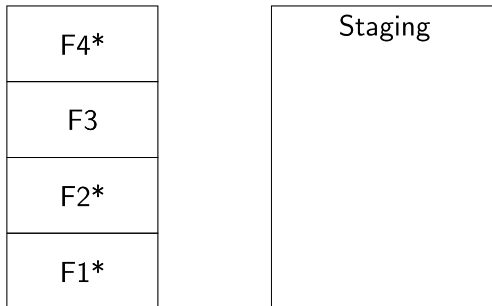
- Add files to the staging environment
- Commit the changes



How to commit

A git commit is a two-stages process.

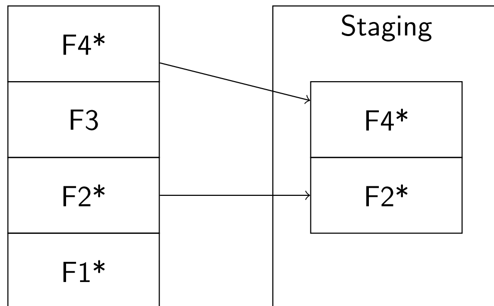
- Add files to the staging environment
- Commit the changes



How to commit

A git commit is a two-stages process.

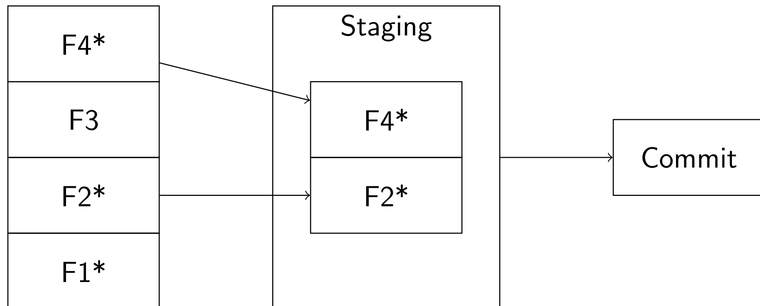
- Add files to the staging environment
- Commit the changes



How to commit

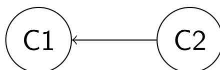
A git commit is a two-stages process.

- Add files to the staging environment
- Commit the changes



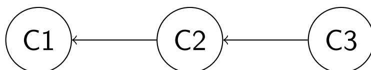
How to commit(2)

- Check modified files
  - `git status`
- Check commits history
  - `git log`
- Add files to the staging environment
  - `git add file/folder`
- Commit the changes
  - `git commit -m "commit message"`



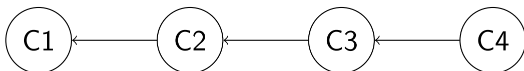
How to commit(2)

- Check modified files
  - `git status`
- Check commits history
  - `git log`
- Add files to the staging environment
  - `git add file/folder`
- Commit the changes
  - `git commit -m "commit message"`

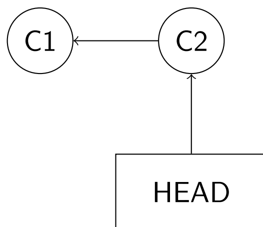


How to commit(2)

- Check modified files
  - `git status`
- Check commits history
  - `git log`
- Add files to the staging environment
  - `git add file/folder`
- Commit the changes
  - `git commit -m "commit message"`

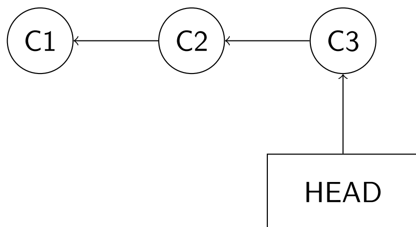


HEAD



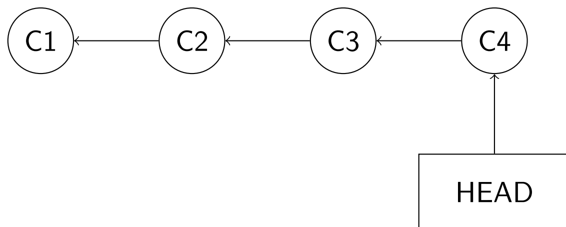
HEAD is a git variable that points to the most recent commit.

HEAD



`HEAD` is a git variable that points to the most recent commit.

HEAD



HEAD is a git variable that points to the most recent commit.

Git superpowers

Your main git superpower is:

- `git reset <NEW_HEAD>`

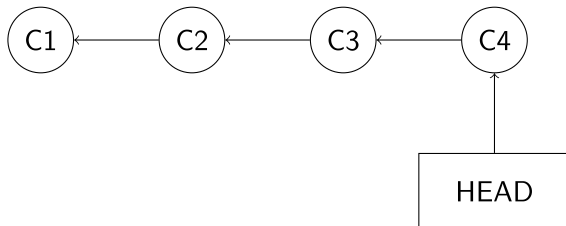
With this command you can change `HEAD` and make it point to a different commit.

- Change HEAD **without** changing the files
  - `git reset <NEW_HEAD>`
- Change HEAD **changing** the files
  - `git reset --hard <NEW_HEAD>`

Git superpowers(2)

For example with:

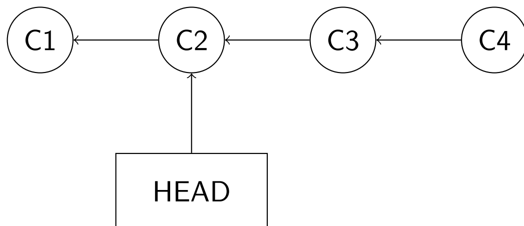
- `git reset --hard C2`



Git superpowers(2)

For example with:

- `git reset --hard C2`



Git superpowers(3)

Don't worry, you are not going to mess it up.

If you `git reset` you will lose in `git log` all the subsequent commit references.

You can retrieve **all HEAD history** with

- `git reflog`

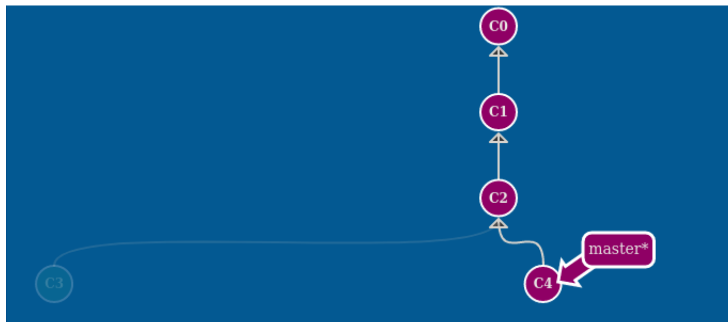
Learn Git Branching

We are going to use a tool called *Learn Git Branching* for the next slides.
This tool allow you to visualize git commands in a graph.
It uses a very simplified subset of git commands:

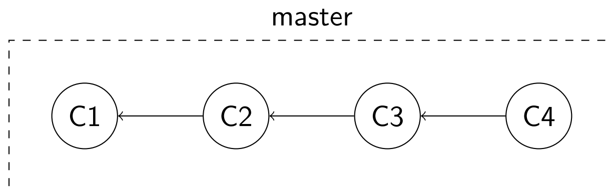
- `git commit`
  - There is no concept of staging environment nor of files
  - You can make a commit without message
- `git reset <hash>`
  - It uses C1, C2, ... CN as hashes
  - It always implies the `--hard` behavior
- `git clone`
  - Treats the repo as it has just been cloned from an identical origin
- `git fakeTeamwork`
  - Creates a new commit in remote origin

Try it yourself!(1)

Exercise 1



Branches



A branch is a (semantically significant) collection of commits.

As a commit represent an atomic unit of modification,
a branch represent a continuous flow of modifications.

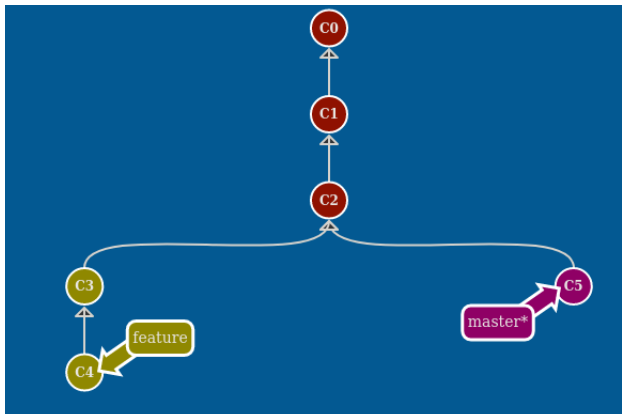
A commit is **always** in a branch.

Branches(2)

- `git branch`
  - Show all the existing branches and the active one (denoted with an asterisk)
- `git branch <branch>`
  - Create a new branch called `<branch>`
- `git checkout <branch>`
  - Switch working on the branch `<branch>`
- `git checkout -b <branch>`
  - Create `<branch>` and switch working on it

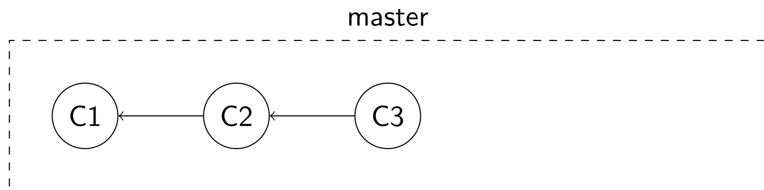
Try it yourself!(2)

Exercise 2



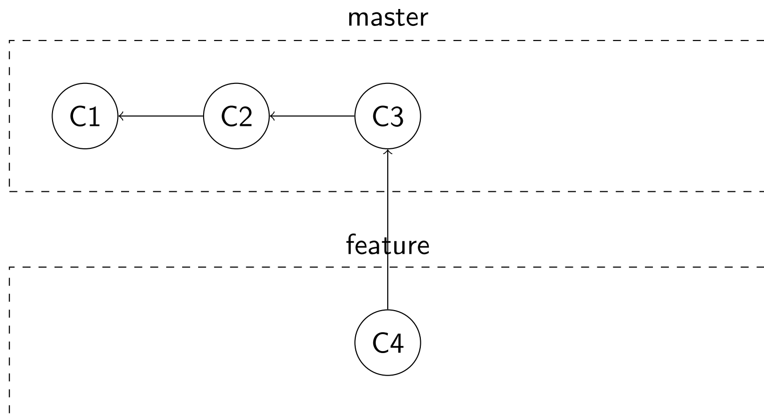
Merge(1)

You can merge different branches (or even single commits).
Most of the times commits has been pushed in both branches.
Don't panic!



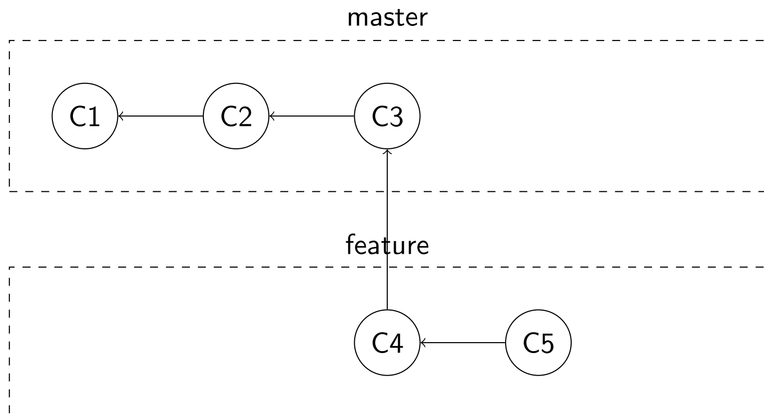
Merge(1)

You can merge different branches (or even single commits).
Most of the times commits has been pushed in both branches.
Don't panic!



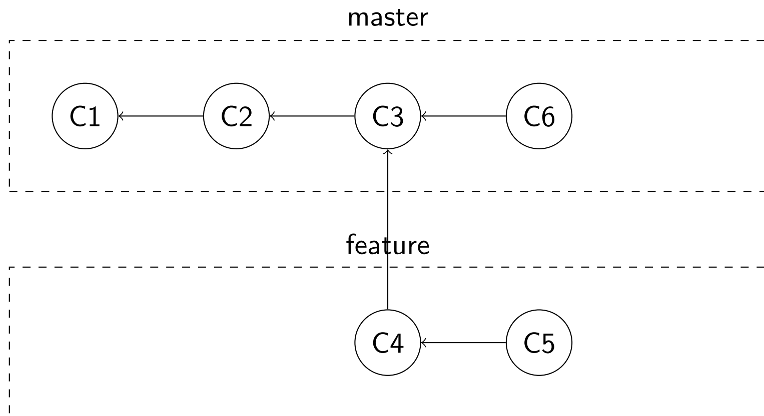
Merge(1)

You can merge different branches (or even single commits).
Most of the times commits has been pushed in both branches.
Don't panic!



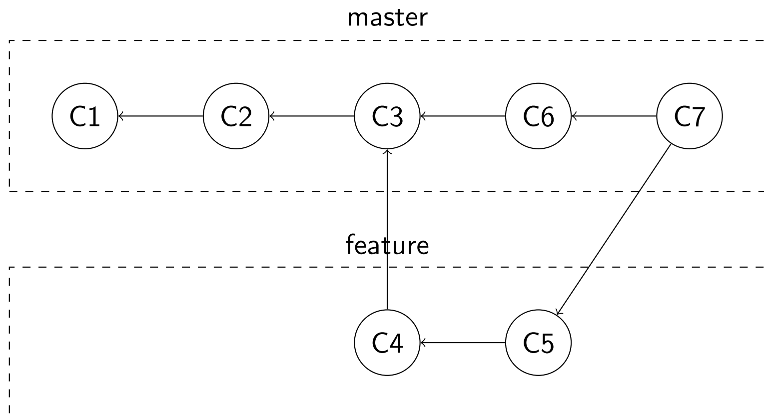
Merge(1)

You can merge different branches (or even single commits).
Most of the times commits has been pushed in both branches.
Don't panic!



Merge(1)

You can merge different branches (or even single commits).
Most of the times commits has been pushed in both branches.
Don't panic!



Merge(2)

Don't worry, git auto-merge is smart.

If the commits changed **different files** the merge is **without conflicts**.

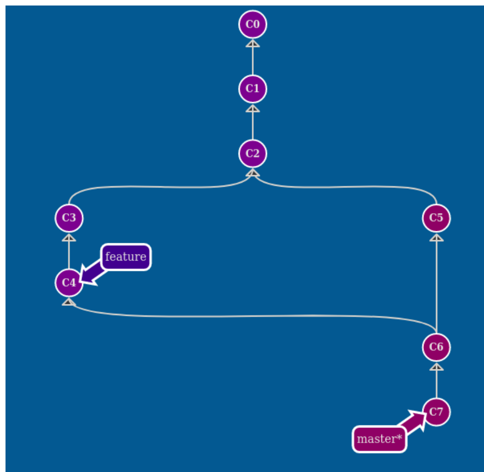
If the commits changed **different sections of the same files** the merge is **without conflicts**.

If the commits changed the **same sections** of the **same files** (at least once) the commit is **with conflict**

- `git merge <branch>`
  - Merges `<branch>` in the active branch
- `git merge --abort`
  - Abort the merge and restore everything as it was before `git merge`

Try it yourself!(3)

Exercise 3



Merge(3)

Handling conflicts is pretty easy.
When a conflict is detected git puts in the place of the conflict:

```
Some non-conflicting text
<<<<<<< HEAD
CONFLICTING PORTION COMING FROM HEAD
=======
CONFLICTING PORTION COMING FROM bugfix
>>>>>>> bugfix
Some other non-conflicting text
```

Is then up to you keep the portions that you want and then `git add` and `git commit` the changes

Remotes

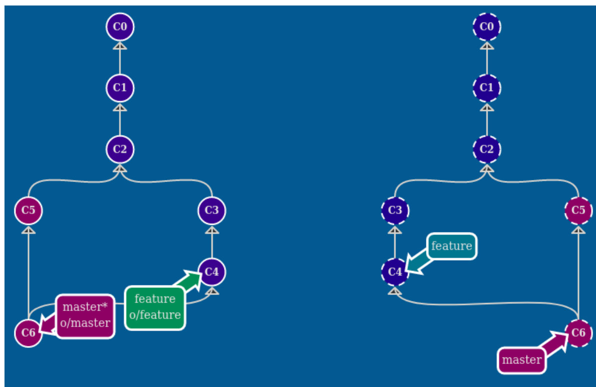
Git is distributed.

A `remote` is a reference to a remote instance of the repository.
The default `remote` is called `origin`.
If you `clone` a repository then `origin` points to the cloned repo.

- `git fetch <remote> <branch>`
  - Fetch the commits from branch `<branch>` of `<remote>` in the local branch `<remote>`/`<branch>`
- `git pull <remote> <branch>`
  - Fetch the commits from branch `<branch>` of `<remote>` and **merge** them in the local `<branch>`
- `git push <remote> <branch>`
  - Push the state of the local `branch` to the `origin` branch `branch`

Try it yourself!(4)

Exercise 4



you can not use `git pull`

Demo time

Lets see some examples in the real world!

Git tags

Tags are a way to point specific points in time (commits) that represent milestones.

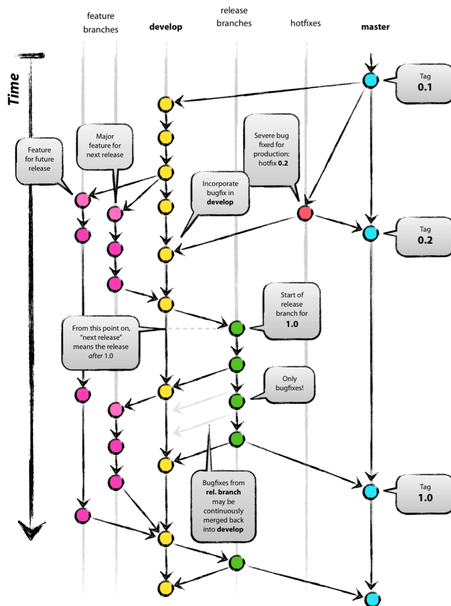
- `git tag <tag> <commit>`
  - Tags `<commit>` with the tag `<tag>`
  - The default commit is HEAD
- `git checkout <tag>`
  - Check out the tag `<tag>`
- `git push <remote> <tag>`
  - Push the (newly created) tag `<tag>` to the remote `<remote>`
- `git push <remote> --tags`
  - Push all the (newly created) tags to the remote `<remote>`

Cherry picking

Cherry picking is the action of merging into HEAD some cherry-picked commits.

- `git cherry-pick <commit>`
  - Applies the changes from commit `<commit>` to HEAD

A successful Git branching model



GitHub

GitHub is git server.

- over 37 Million Users
- over 100 Million Repositories
- Largest source code host in the world
- Home of millions of free and open source projects

GitHub forks

In GitHub there is the concept of *fork*.
A fork is a clone of someone else's repository into yours repositories.

A fork is not a git clone

After a fork you **own** an exact copy of a repository.
Copy of which you and only you are the administrator.

GitHub Pull Requests

If you want to ask for the integration of your changes in the forked repository you have to open a **Pull Request**.

In the pull request you have to specify the destination branch (original repository) and the source branch (your repository)

That's all folks!

You can find the slides PDF as well as their LaTeX source code on GitHub.

https://github.com/galatolofederico/git-very-informal-introduction