Git workshop

Ángel de Vicente

POLMAG project (http://research.iac.es/proyecto/polmag/ (http://research.iac.es/proyecto/polmag/))

E-mail: *angel.de.vicente@iac.es*

May 12-13, SUNDIAL

Index

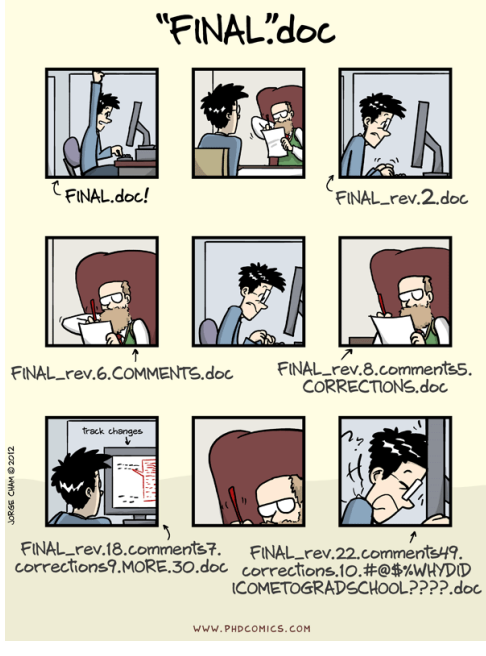
1. Git basics

I'm assumming that you are familiar with the following concepts:

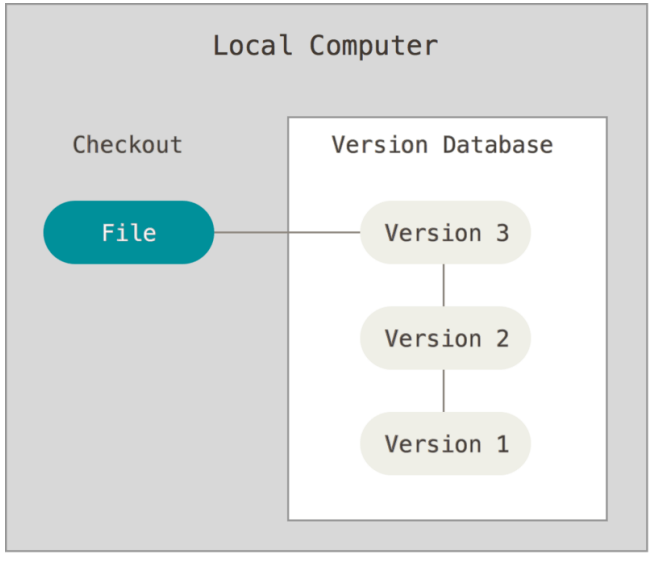
- What is version control?
- What do we version control (text vs. binary files)
- What is a version control system
- You have used a version control system directly or indirectly (e.g. Dropbox, Overleaf, etc.)
- Version control ideas: record/playback, branching, merging
- Benefits of version control systems

Types of Version Control Systems

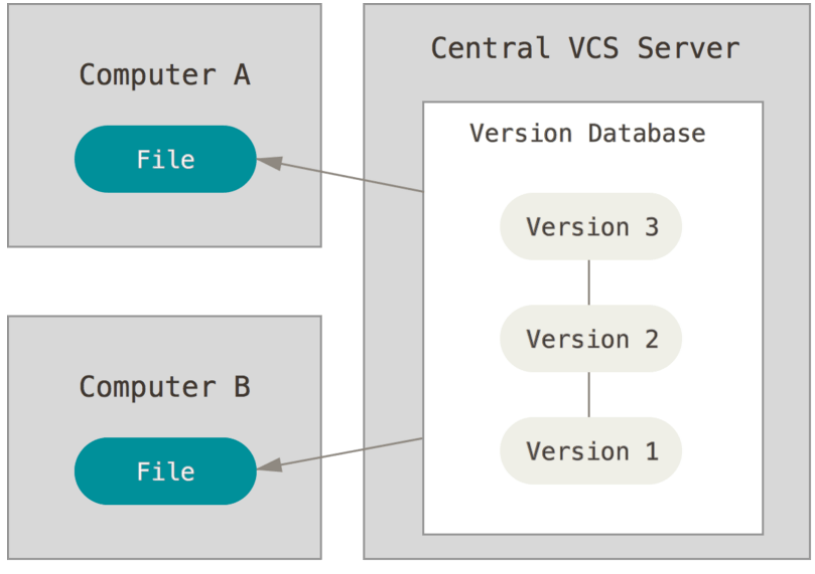
- No VCS (*don't be the one doing this!*)



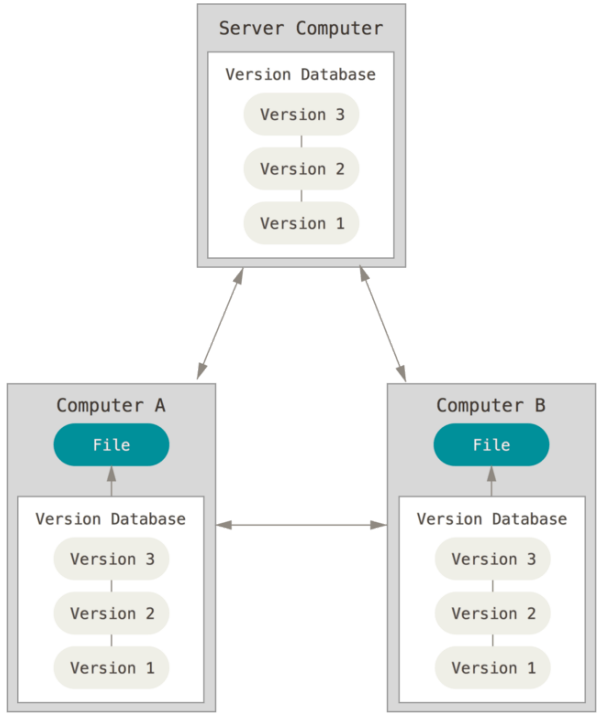
- Local VCS (e.g. RCS)



- Centralized VCS (e.g. `Subversion`)



- Distributed VCS (e.g. `Git`)



I'm assumming that:

- You are *familiar* with Git
- About Git: https://git-scm.com/ (https://git-scm.com/) (downloads, documentation, etc.)
- You have *git* installed in your computer
- You have a *GitHub* account (https://github.com/ (https://github.com/))
- You are using GNU/Linux (small variations if using Mac OS X or Windows)
- Much more than what we'll do today:
    - Pro Git book: https://git-scm.com/book/en/v2 (https://git-scm.com/book/en/v2) (most images taken from here)

Setting up git

- *git config* (details at: https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup (https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup))
- Configuration variables stored in three different places:
    - */etc/gitconfig* file (*--system* option)
    - *~/.gitconfig* or *~/.config/git/config* file (*--global* option)
    - *config* file in the Git directory (*--local* option)

Setting up git (2)

- Basic settings

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

- Many other options, e.g:

```
$ git config --global core.editor emacs
$ git config --global color.ui "auto"
```

Sample settings

In [1]:
```
! git config --list ## --show-origin
```

```
user.email=angel.de.vicente@iac.es
user.name=Angel de Vicente
color.ui=true
color.status=auto
color.branch=auto
credential.helper=cache --timeout=28800
diff.jupyternotebook.command=git-nbdiffdriver diff
merge.jupyternotebook.driver=git-nbmergedriver merge %O %A %B %L %P
merge.jupyternotebook.name=jupyter notebook merge driver
difftool.nbdime.cmd=git-nbdifftool diff "$LOCAL" "$REMOTE" "$BASE"
difftool.prompt=false
mergetool.nbdime.cmd=git-nbmergetool merge "$BASE" "$LOCAL" "$REMOTE" "$MERGE
D"
mergetool.prompt=false
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=https://github.com/angel-devicente/git-workshop.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
remote.pushdefault=origin
```

Getting help

In [2]:
```
! git --version
```

```
git version 2.25.1
```

In [3]:
```
! git --help | head -n 25
```

```
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone             Clone a repository into a new directory
   init              Create an empty Git repository or reinitialize an existin
g one

work on the current change (see also: git help everyday)
   add               Add file contents to the index
   mv                Move or rename a file, a directory, or a symlink
   restore           Restore working tree files
   rm                Remove files from the working tree and from the index
   sparse-checkout   Initialize and modify the sparse-checkout

examine the history and state (see also: git help revisions)
   bisect            Use binary search to find the commit that introduced a bu
g
   diff              Show changes between commits, commit and working tree, et
c
   grep              Print lines matching a pattern
   log               Show commit logs
   show              Show various types of objects
```

In [4]:
```
! git status -h
```

```
usage: git status [<options>] [--] <pathspec>...

    -v, --verbose         be verbose
    -s, --short           show status concisely
    -b, --branch          show branch information
    --show-stash          show stash information
    --ahead-behind        compute full ahead/behind values
    --porcelain[=<version>]
                          machine-readable output
    --long                show status in long format (default)
    -z, --null            terminate entries with NUL
    -u, --untracked-files[=<mode>]
                          show untracked files, optional modes: all, normal, n
o. (Default: all)
    --ignored[=<mode>]    show ignored files, optional modes: traditional, mat
ching, no. (Default: traditional)
    --ignore-submodules[=<when>]
                          ignore changes to submodules, optional when: all, di
rty, untracked. (Default: all)
    --column[=<style>]    list untracked files in columns
    --no-renames          do not detect renames
    -M, --find-renames[=<n>]
                          detect renames, optionally set similarity index
```

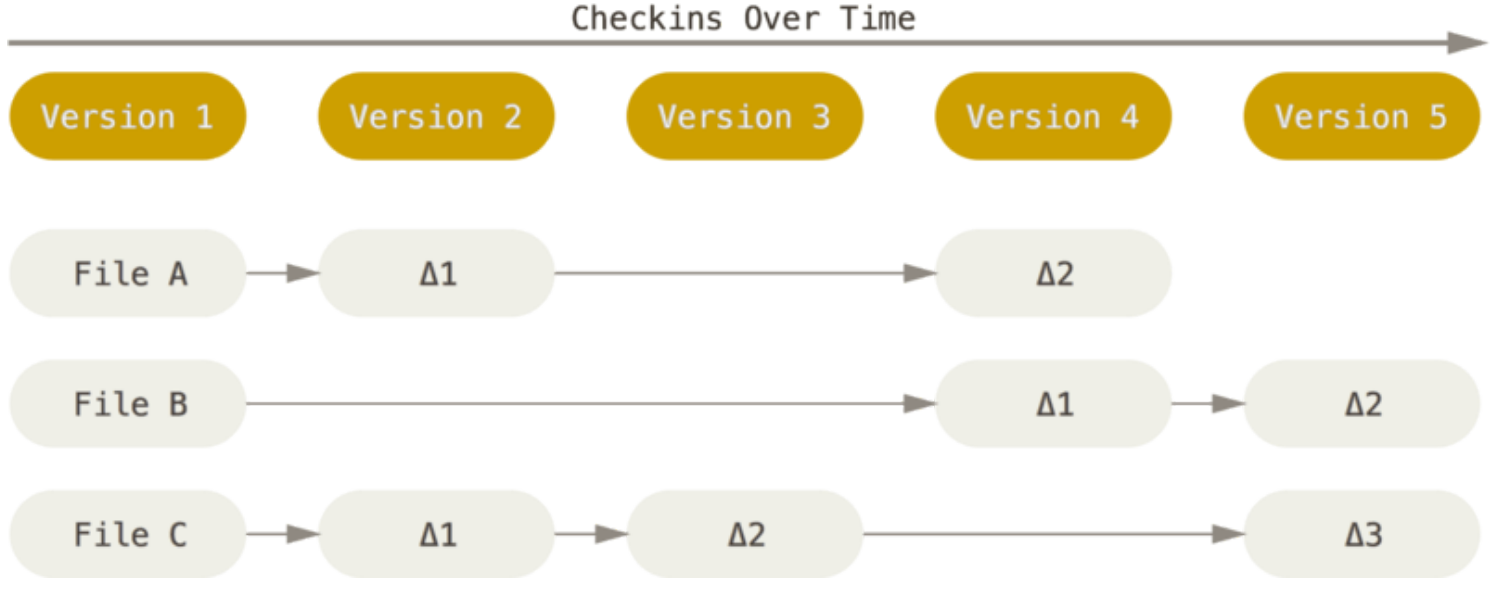
In [5]:
```
! git status --help | head -n 22
```

```
GIT-STATUS(1)                     Git Manual                     GIT-STATUS(1)

NAME
       git-status - Show the working tree status

SYNOPSIS
       git status [<options>...] [--] [<pathspec>...]

DESCRIPTION
       Displays paths that have differences between the index file and the
       current HEAD commit, paths that have differences between the working
       tree and the index file, and paths in the working tree that are not
       tracked by Git (and are not ignored by gitignore(5)). The first are
       what you would commit by running git commit; the second and third are
       what you could commit by running git add before running git commit.

OPTIONS
       -s, --short
           Give the output in the short-format.

       -b, --branch
           Show the branch and tracking info even in short-format.
```

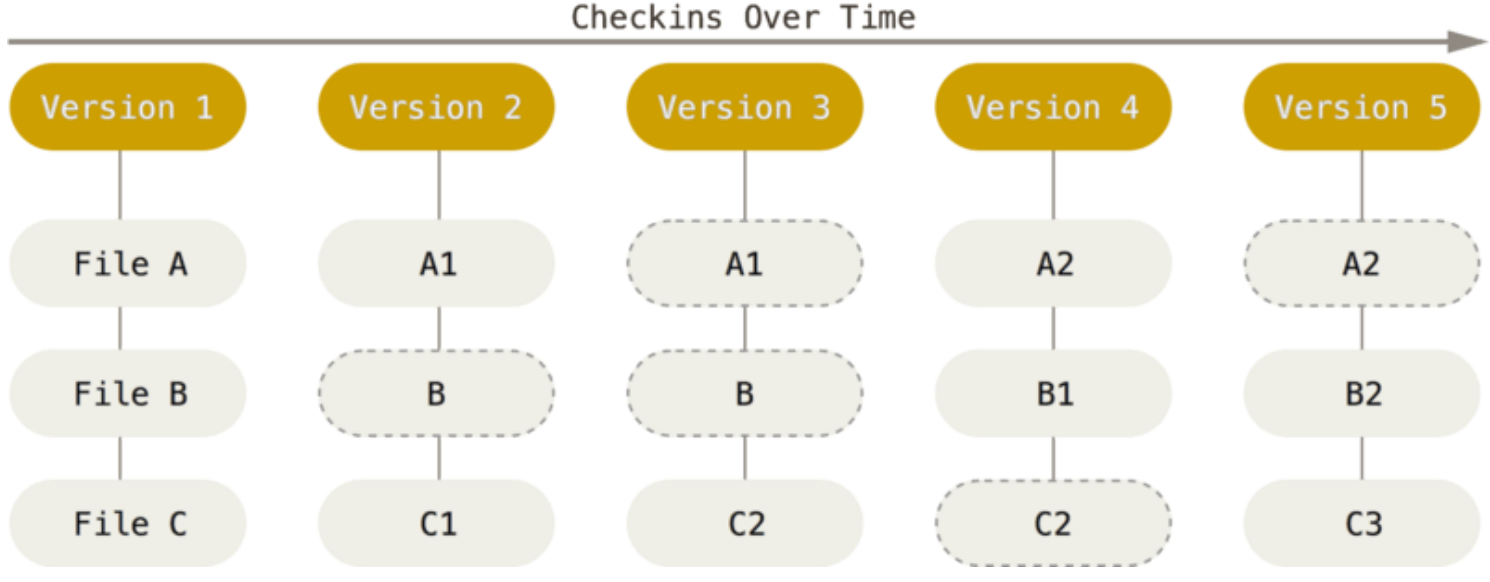
Main ideas in Git

- Nearly all operations are local
    - *.git* directory keeps all history
    - very little that you cannot do when offline
- Integrity
    - Everything checksummed before stored
    - Hashes: *24b9da6552252987aa493b52f8696cd6d3b00373*
- Snapshots, not differences

- Delta-based VCS

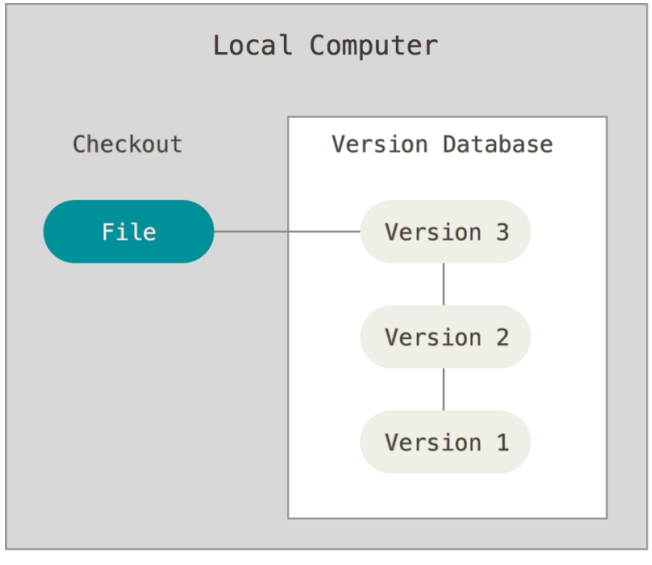


- Snapshots (Git)



2. Git basic usage

Using git as a local VCS



Create a repository

git init

In [6]:
```
%%bash

cd ../Demos/Repos
rm -rf First_Demo ; mkdir First_Demo ; cd First_Demo

git init -q
ls -al
```

```
total 12
drwxr-xr-x 3 angelv angelv 4096 May 12 10:43 .
drwxr-xr-x 6 angelv angelv 4096 May 12 10:43 ..
drwxr-xr-x 7 angelv angelv 4096 May 12 10:43 .git
```

- All repository information goes to .git directory (DO NOT EDIT by hand)

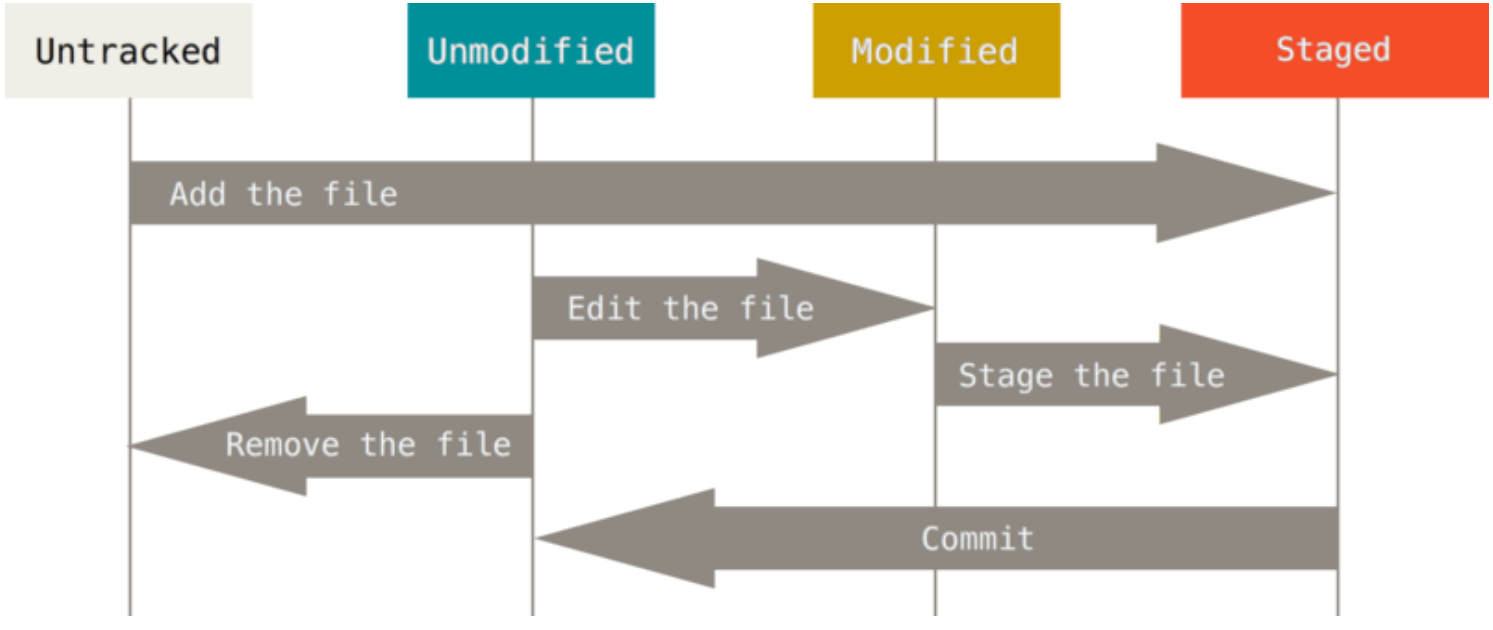
In [7]:
```
%%bash

cd ../Demos/Repos
ls First_Demo/.git
```

```
branches
config
description
HEAD
hooks
info
objects
refs
```

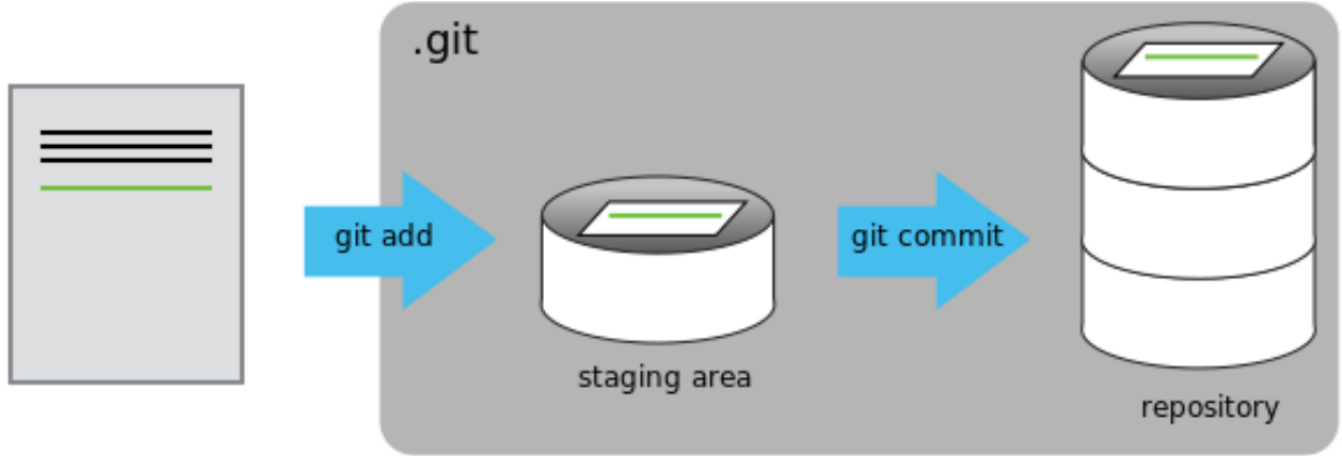
Recording changes

- `git status`
- `git add`
- `git commit`
- `git rm`
- `git mv`

- Tracking changes



- Commiting changes



Recording changes demo

demo-1-1.cast (https://asciinema.org/a/Rff5PnrBC79FMI7ThpbWBLX8D?autoplay=1&cols=180&rows=40)

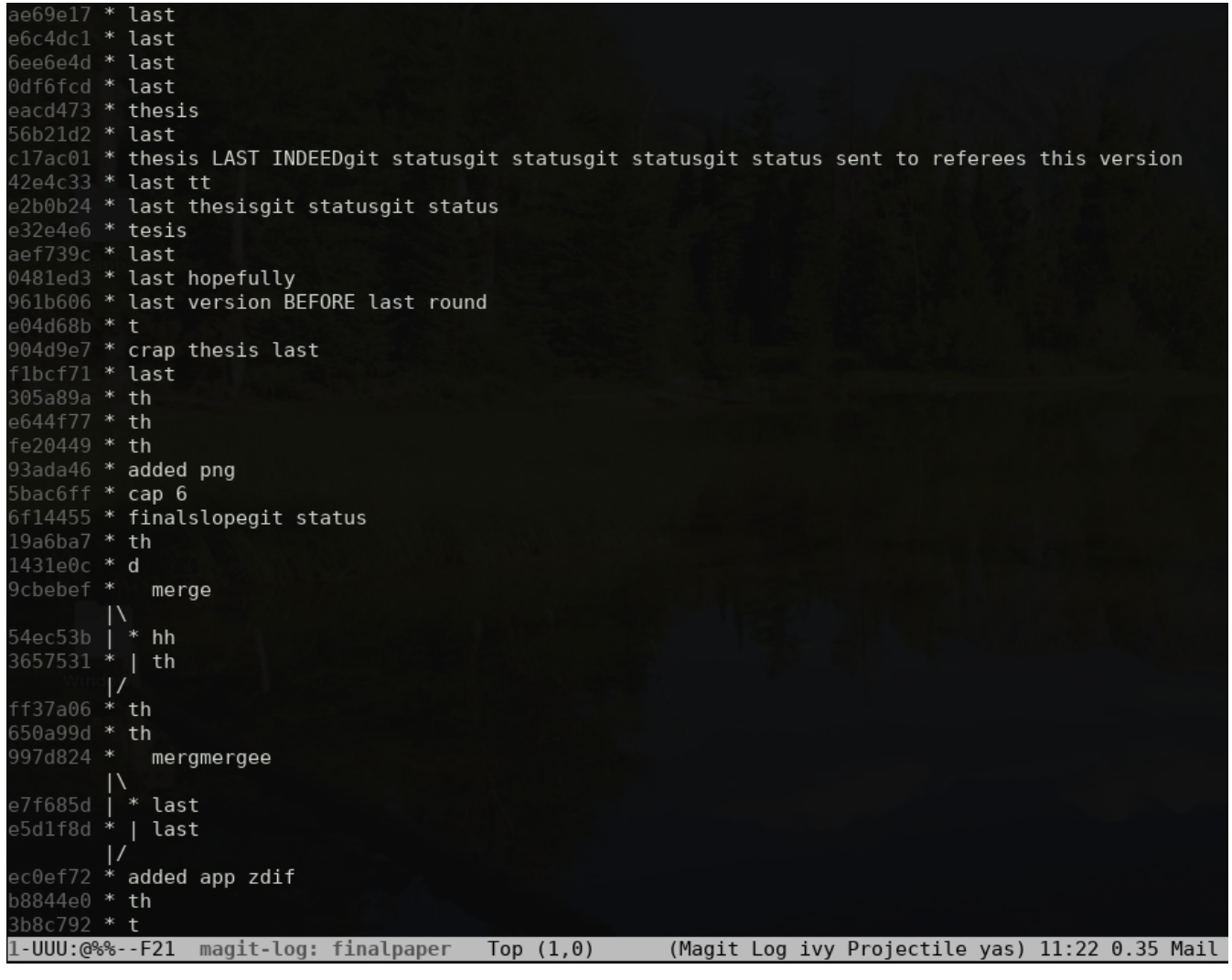
Viewing history

- `git log`
- `git diff`

Viewing history demo

demo-1-2.cast (https://asciinema.org/a/zHPylAmuKN3TWmGhQJiPOidZd?autoplay=1&cols=180&rows=40)

- Commit messages are important (*don't do this*)



Undoing things

- `git commit --amend`
- `git restore --staged CONTRIBUTING.md`
- `git restore README.md`

Undoing things demo

demo-1-3.cast (https://asciinema.org/a/Sq5xBDSbbbdJ1AxdZmz755xaJ?autoplay=1&cols=180&rows=40)

- In older versions of `git`, instead of `git restore` you would use:
  - `git reset HEAD ...`
  - and `git checkout ...`

    (see examples in https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things (https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things))

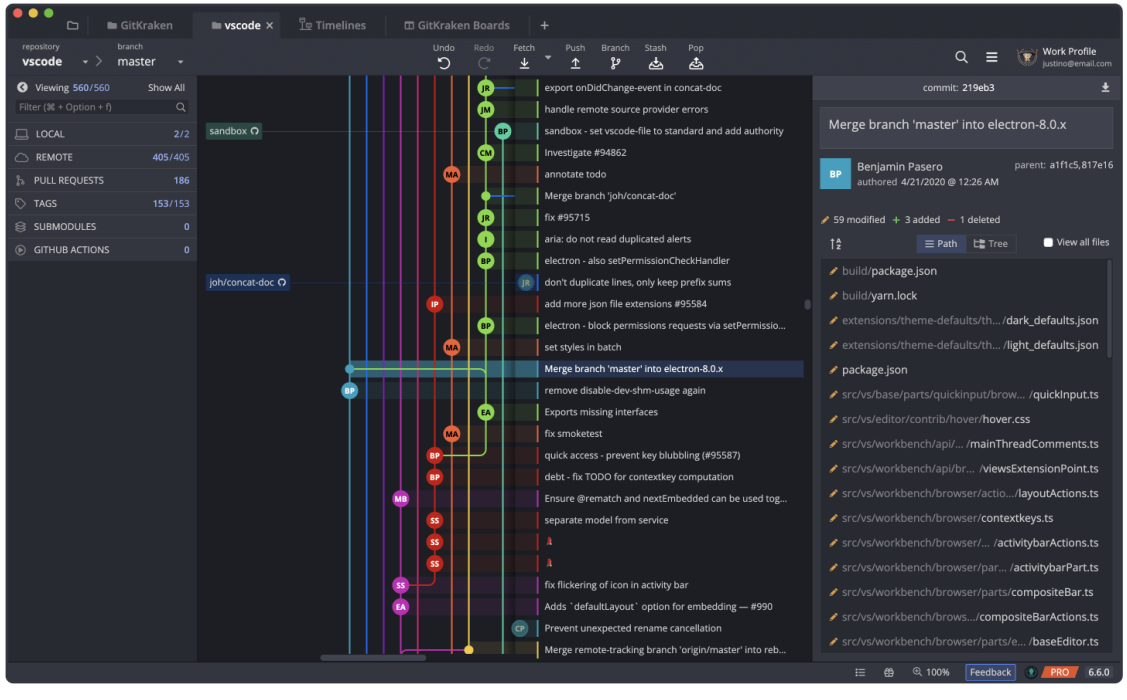
Practice time

- Get some repository with plenty of history (we'll see `git clone` later)
    - `git clone https://github.com/jlord/git-it-electron.git`
    - cd git-it-electron
- How many commits in total are made by Brendan Forster?
- Find the date and the hash of the commit by Nikolai Plath where "GitHub" is mentioned in the commit message.
- Undo the change to the `README.md` file made by him.
    - *Hints:*
        - to find what was changed in a commit you can use `git show`. Get help for this command.
        - to revert changes do it manually or check the command `git revert`
- Add files `BUGS.md` and `FIXES.md` and commit with message "Fixes"
- Add some content to `BUGS.md` and amend the commit with message "Fixes and bugs"
- Rename `BUGS.md` to `BUGS_FIXES.md` and delete `FIXES.md` and commit again (amending)

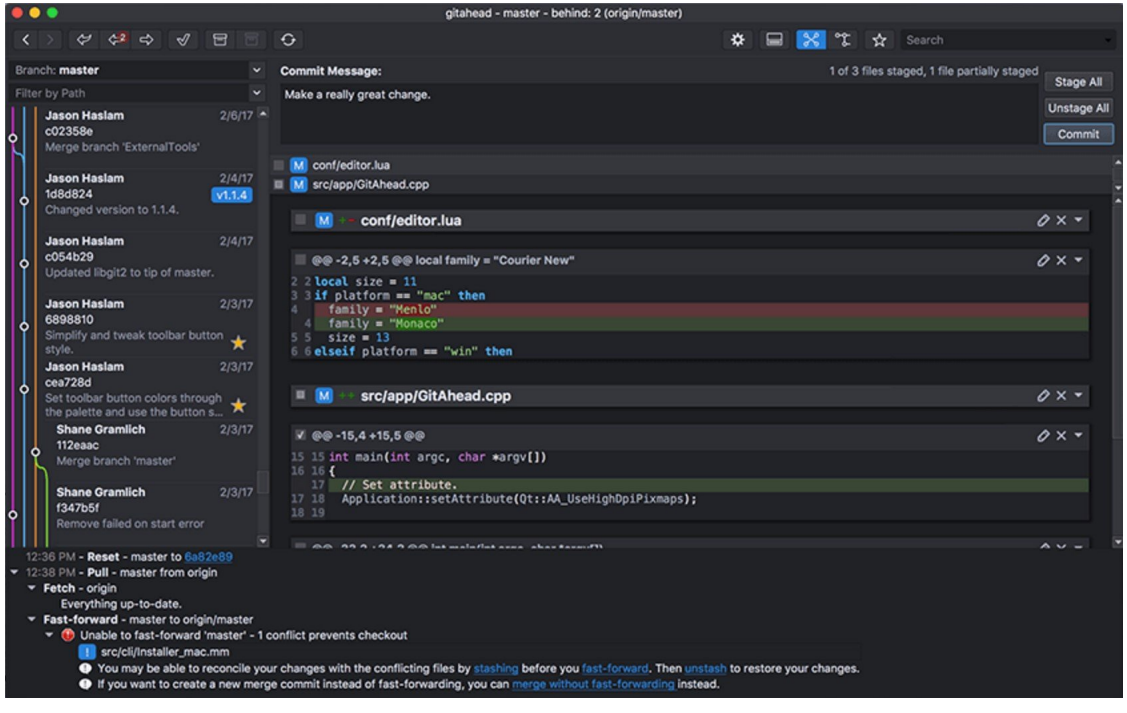
Command-line vs GUIs

- Command-line lets you:
    - access *every* aspect of Git
    - can be automated in scripts
    - 'lingua franca'
- GUIs
    - the entry barrier for users is lower
    - can group common usage patterns
    - help with advanced options
    - choose your poison: GitKraken, GitAhead, Magit, ...
        - https://git-scm.com/download/gui/linux (https://git-scm.com/download/gui/linux)

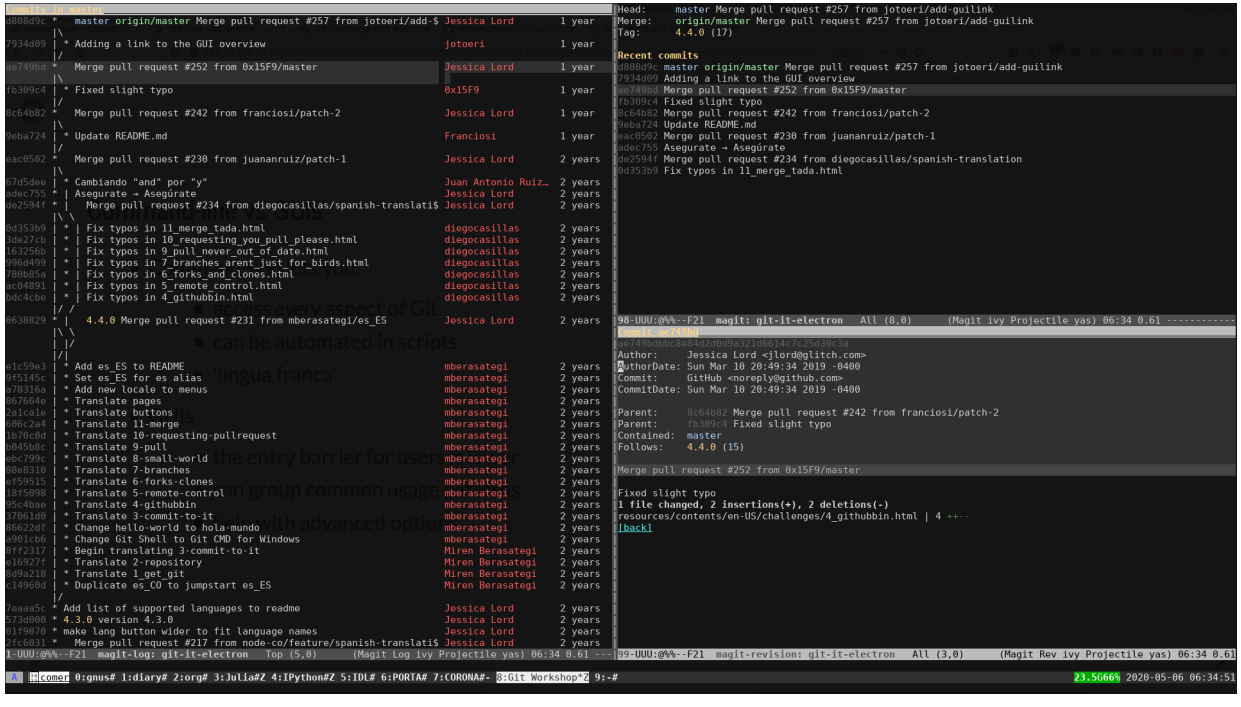
- GitKraken [https://www.gitkraken.com/ (https://www.gitkraken.com/)](https://www.gitkraken.com/)



- GitAhead https://gitahead.github.io/gitahead.com/ (https://gitahead.github.io/gitahead.com/)



- Magit (for Emacs) [https://magit.vc/ (https://magit.vc/)](https://magit.vc/)



Live demo with Magit & GitAhead

- (see usefulness of hunks)

3. Working with remotes

Remote repositories

- Remote repositories are versions of your project that are hosted:
    - on the Internet
    - or more generally in the network somewhere (even in your local machine)
- Crucial for collaborating with others:
    - pushing and pulling data from remotes
- GitHub, GitLab, Bitbucket, etc. are on-line remotes with:
    - extras for collaboration
    - features for code development (e.g. CI/CD)

Basic remotes commands

- `git remote ...`
  - `-v` add show `rename` `remove`
- `git ...`
  - `clone`
  - `pull`
  - `push`
  - `fetch`
  - `merge`

In [9]:
```
%%bash

cd ../Demos/Repos/git-it-electron
git remote -v
```

```
origin  https://github.com/jlord/git-it-electron.git (fetch)
origin  https://github.com/jlord/git-it-electron.git (push)
```

In [10]:
```
%%bash

cd ../Demos/Repos/git-it-electron
git remote show origin
```

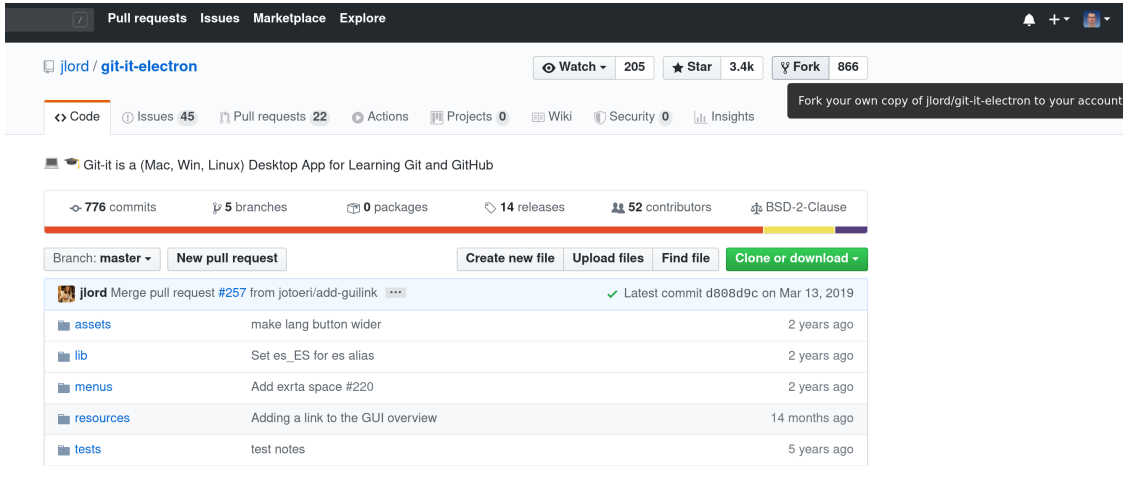
```
* remote origin
  Fetch URL: https://github.com/jlord/git-it-electron.git
  Push  URL: https://github.com/jlord/git-it-electron.git
  HEAD branch: master
  Remote branches:
    IUTInfoAix-M2105-french-translation      tracked
    dependabot/npm_and_yarn/handlebars-4.3.0 tracked
    fix-menu                                 tracked
    lang-tweaks                              tracked
    master                                   tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Getting started with GitHub (similar for GitLab, Bitbucket, etc.)

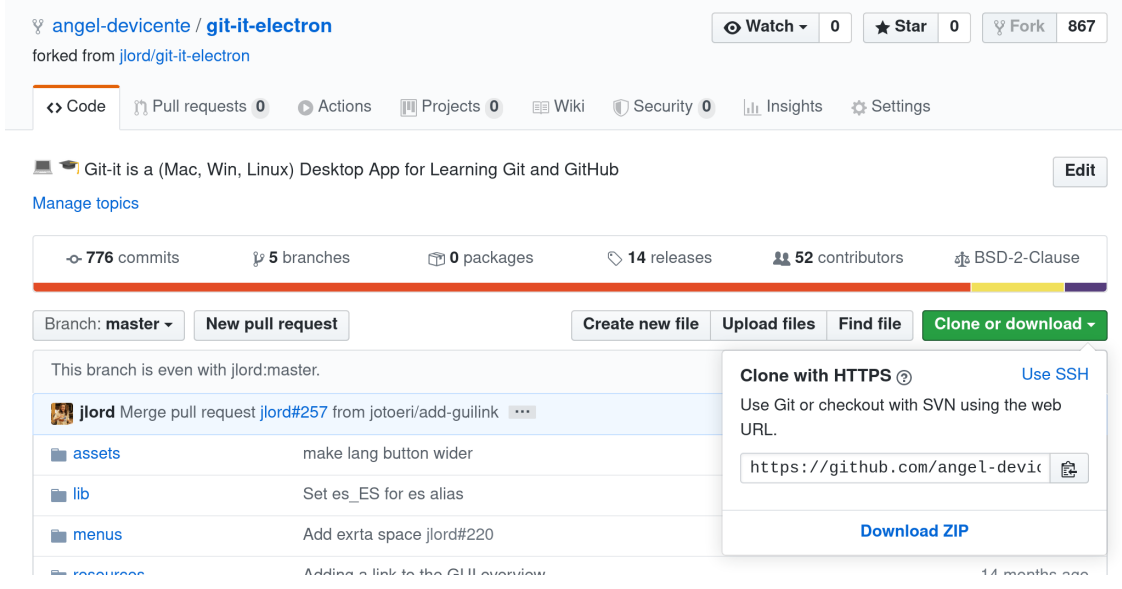
- Login to your GitHub account
- Two ways to get a new repository:
    - *Fork* an existing repository
    - Create a new repository

Fork a repository from GitHub into your account

- Fork the repository *git-it-electron* (https://github.com/jlord/git-it-electron (https://github.com/jlord/git-it-electron))



Clone your newly *forked* repository

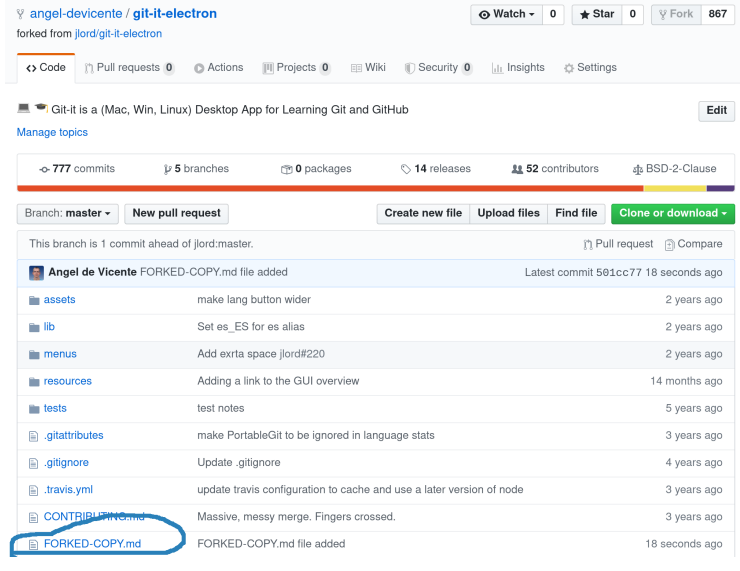


In [11]:
```bash
%%bash

cd ../Demos/Repos

# USE YOUR forked repository, not mine!
#
# git clone https://github.com/angel-devicente/git-it-electron.git
# fatal: destination path 'git-it-electron' already exists and is not an empty directory.

if [ ! -d git-it-cp1 ] ; then
    git clone https://github.com/angel-devicente/git-it-electron.git git-it-cp1
fi
```

In [12]:
```bash
%%bash

cd ../Demos/Repos/git-it-cp1

git remote -v
```

```
origin  https://github.com/angel-devicente/git-it-electron.git (fetch)
origin  https://github.com/angel-devicente/git-it-electron.git (push)
```

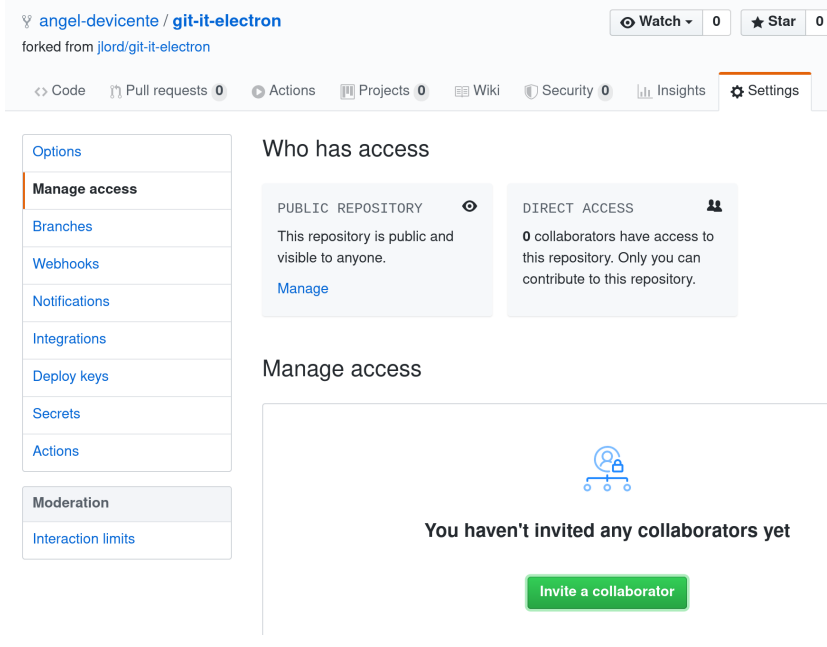
- Create a new file in the git-it-cp1 repository and *push* it to GitHub



Collaborating in GitHub

- The commands `fetch`/`merge` and `pull` are useful when collaborating with 'others'
    - another collaborator has `push`ed changes to the repository that you want to get in your local copy
    - you want to synchronize your own changes in different computers
    - you want to get changes from another repository branch (*we will see this later on*)

Granting collaborators access in GitHub



Let's simplify and collaborate with ourselves

- Clone a second copy of `git-it-electron`, now to directory `git-it-cp2`

In [13]:
```
%%bash

cd ../Demos/Repos
if [ ! -d git-it-cp2 ] ; then
    git clone https://github.com/angel-devicente/git-it-electron.git git-it-cp2
fi
```

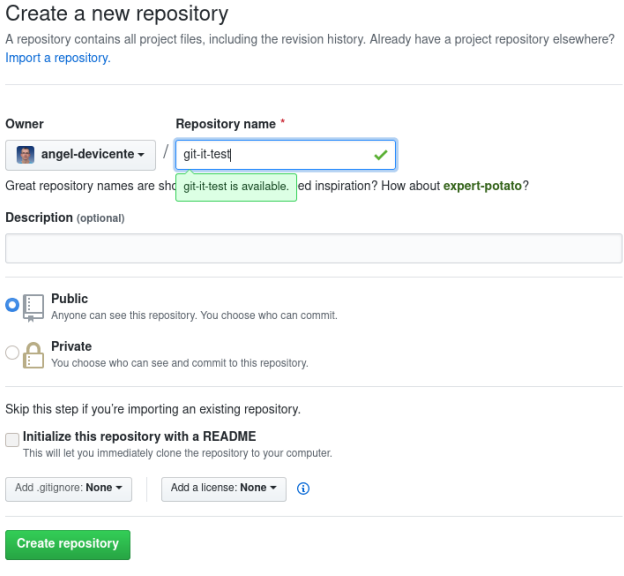
Pull and fetch/merge demo

demo-1-4.cast (https://asciinema.org/a/9UTZj3TVwXm6jpftZ2ZcOdk5d?autoplay=1&cols=180&rows=40)

- As *cp1* I make some changes and `push` to GitHub. Later, as *cp2* we `pull` those changes.
- Then, as *cp2* I make some changes and `push` to GitHub. Later, as *cp1* I:
    - first `fetch` the changes, so I can review them
    - when happy with introducing those changes I can `merge` them to my repository. (*merge* in this simple case is basically just *absorb* all the changes)
- Did you notice *origin/master*? *master* is **tracking** *origin/master*. Confusing? It will become clear later when talking about branches. Hold on!

Create a new repository in GitHub (same idea for GitLab, Bitbucket, etc.)

- Create an empty repository in GitHub ...



- ... and then follow the instructions given by GitHub (reproduced below).
    - all steps should be familiar
    - but why *git push -u origin master*?
    - what happens if you just use *git push* as in the previous demo?
        - ... we need to learn more about branches

```
echo "# git-it-test" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/angel-devicente/git-it-test.git
git push -u origin master
```

4. Branches

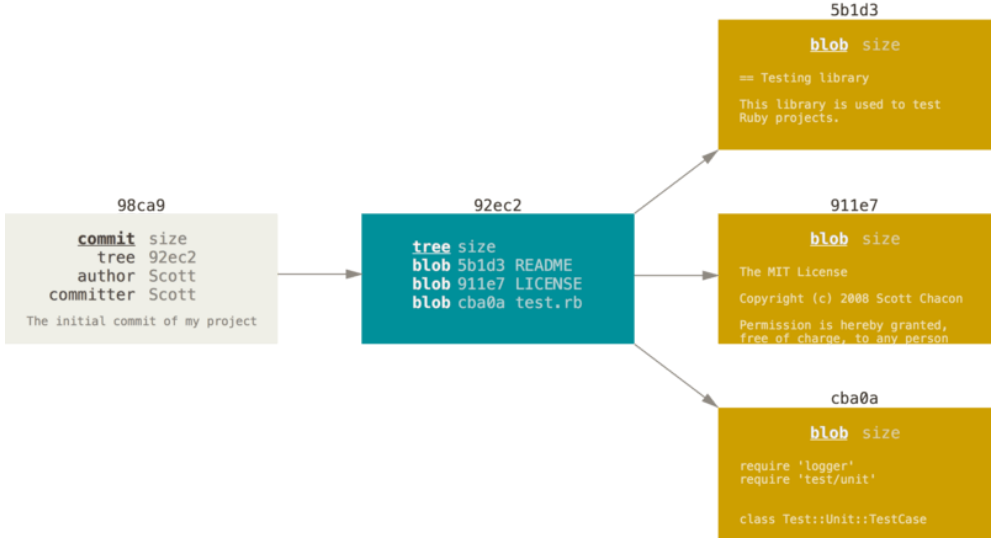
Branches

- A lot of the power (and some confusion) in Git comes from branches
- Branches are the *killer* feature of Git, very lightweight compared to other VCS
- This encourages workflows that create branches and merge them very often.

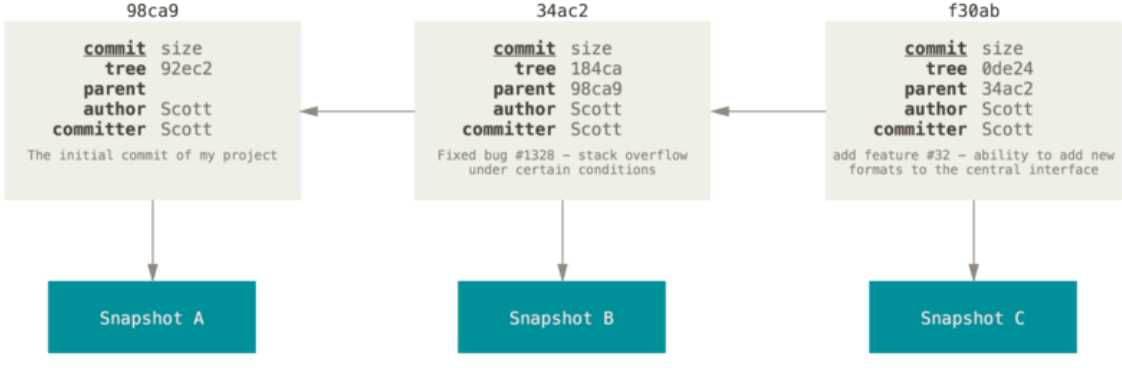
So, let's try and see some of the Git internals to understand what branches are...

(see https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell (https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell))

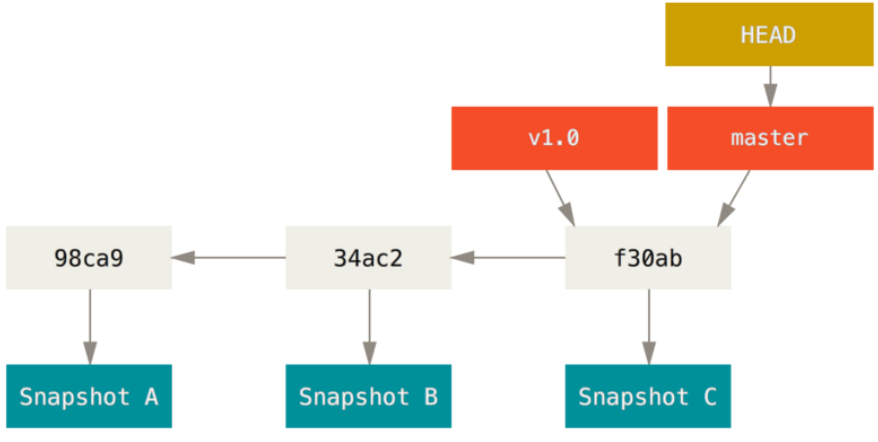
- A *commit* points to a *tree* object, and this to *blobs*
    - For us the important part is just the *commit*, which points to a *snapshot* of our work



- Creating more *commit*s, keeps a linked list
    - (which is basically what you see when you do `git log`)

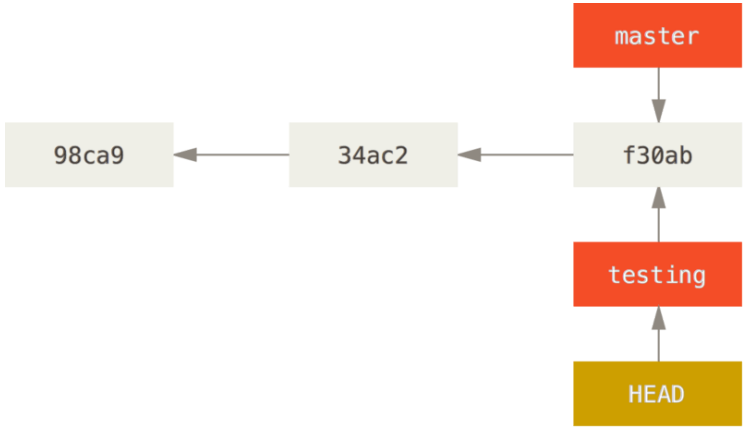


- A *branch* is just a pointer to one of these *commits*
    - *master* is created when you do `git init`, but it is no special
    - *HEAD* points to the branch we are working on

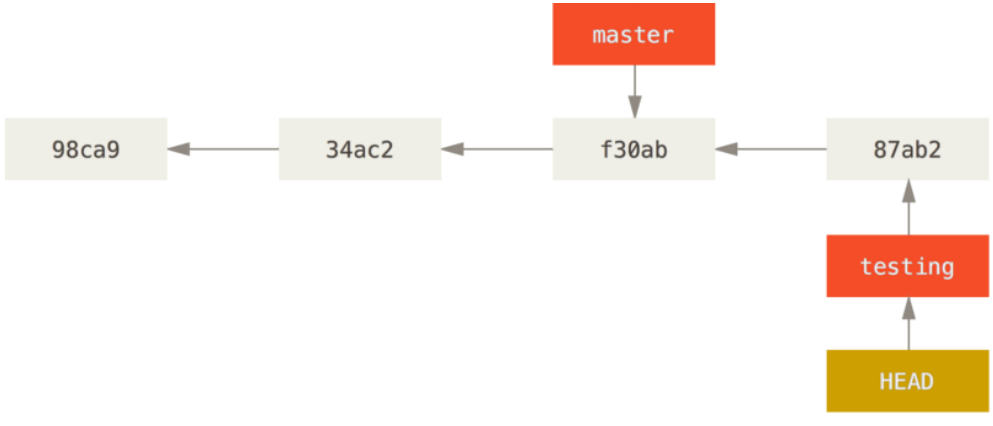


- Creating a branch
    - If we are working in *master* and we do `git branch testing`
        - *testing* branch is created, but we are still working on *master*
        - so you can see *HEAD* is still pointing to *master*

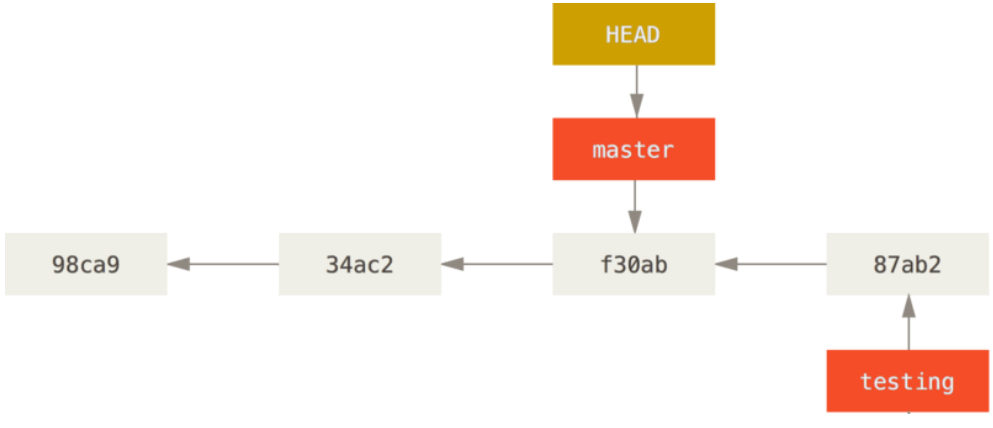
- To work with another branch
  - `git checkout testing` makes *HEAD* point to testing
    - when you switch branches in Git:
      - **files in your working directory will change**.
      - **if Git cannot do it cleanly, it will not let you switch at all**.



- Committing to *testing*
    - makes *testing* branch move forward
    - but *master* branch still points to the same commit



- Changing branches
  - `git checkout master` (if *working tree* is *dirty* we won't be able to change branches)



Practice time

- In *git-it-cp1*:
    - create a new branch *test_branch1* and change to it
    - add a line to a file and commit changes
    - change back to branch *master* and verify the added line is not present in this branch
    - use `git log` to view the history of all branches (should be able to see in first position both branches: *master* and *test_branch1*, plus *HEAD*)

demo-1-5.cast (https://asciinema.org/a/0du1PPZviaW6EoQ7DSATEiAqe?autoplay=1&cols=180&rows=40)

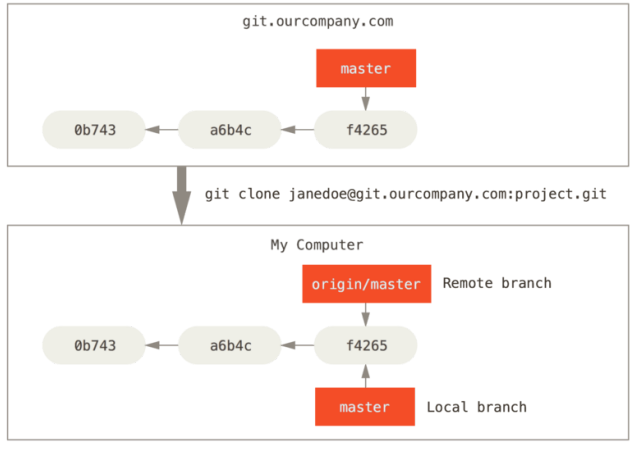
Remote Branches

If you read https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches (https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches), you get all the details, but it can be a bit confusing. Most of the time, you just need three concepsts:

- Remote branch: a branch in a **remote** repository, for example:
    - GitHub,
    - a repository in another PC of yours,
    - another directory in the same machine
- Remote-tracking branch: a **local** *bookmark* pointing to a remote branch
    - can become out-of-date if the remote branch gets new commits
- Tracking branch: a **local** branch, linking the *local* branch to the *remote* branch.

Let's see some examples

- `git clone` will take the remote repository in *git.ourcompany.com* and copy it to your local repository.
    - it will create remote-tracking branches (here *origin/master*)
    - it will create and checkout a *local* tracking branch named *master*, (Only one local branch is created, linked to the currently active branch in the remote repository).

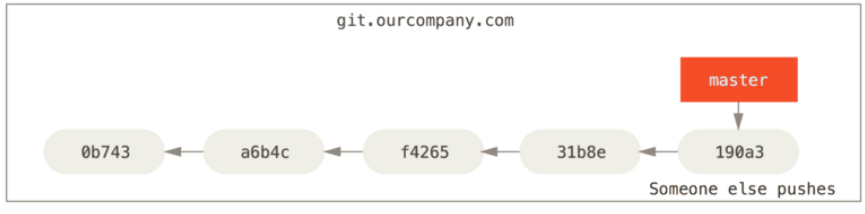


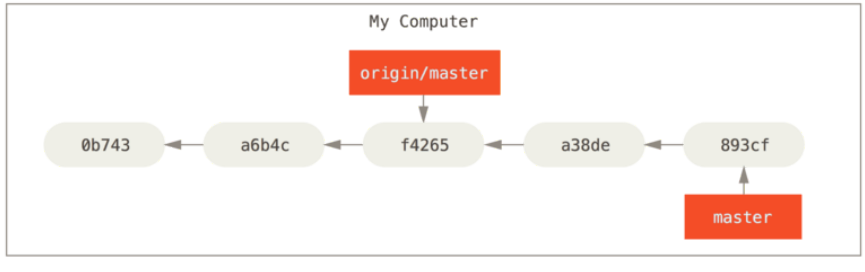
In [14]:

```
%%bash

cd ../Demos/Repos/git-it-cp2
git remote -vv
echo
git branch --all
echo
git branch -vv
```

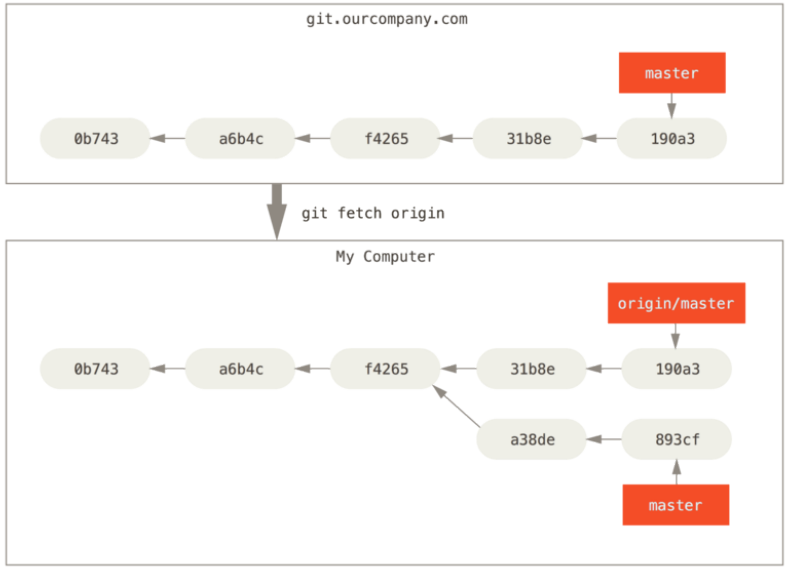
```
origin  https://github.com/angel-devicente/git-it-electron.git (fetch)
origin  https://github.com/angel-devicente/git-it-electron.git (push)

* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/IUTInfoAix-M2105-french-translation
  remotes/origin/dependabot/npm_and_yarn/handlebars-4.3.0
  remotes/origin/fix-menu
  remotes/origin/lang-tweaks
  remotes/origin/master

* master b992ccb [origin/master] Added FORKED-COPY.md
```

- Remote and local repositories can become unsynchronized
    - Here there were commits in the local branch and the remote branch
    - Note that the remote-tracking branch *origin/master* doesn't move (until you use `git fetch` your view of the remote repository is out-of-date)

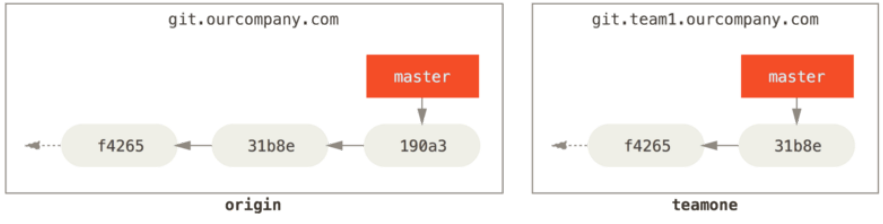




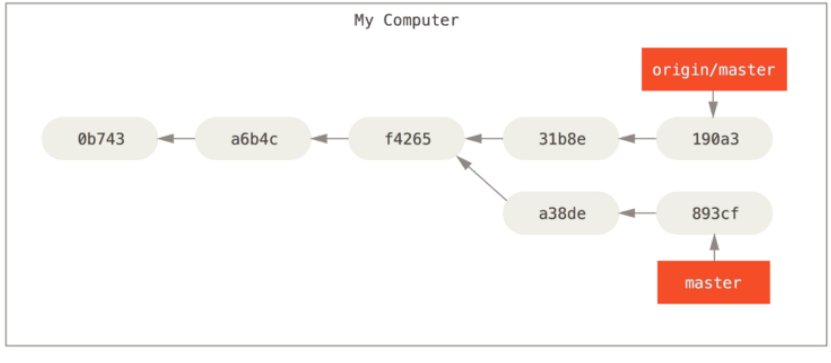
- `git fetch` will synchronize your remote-tracking branch
    - at this point *origin/master* and *master* have diverged (will have to use `git merge` or similar)



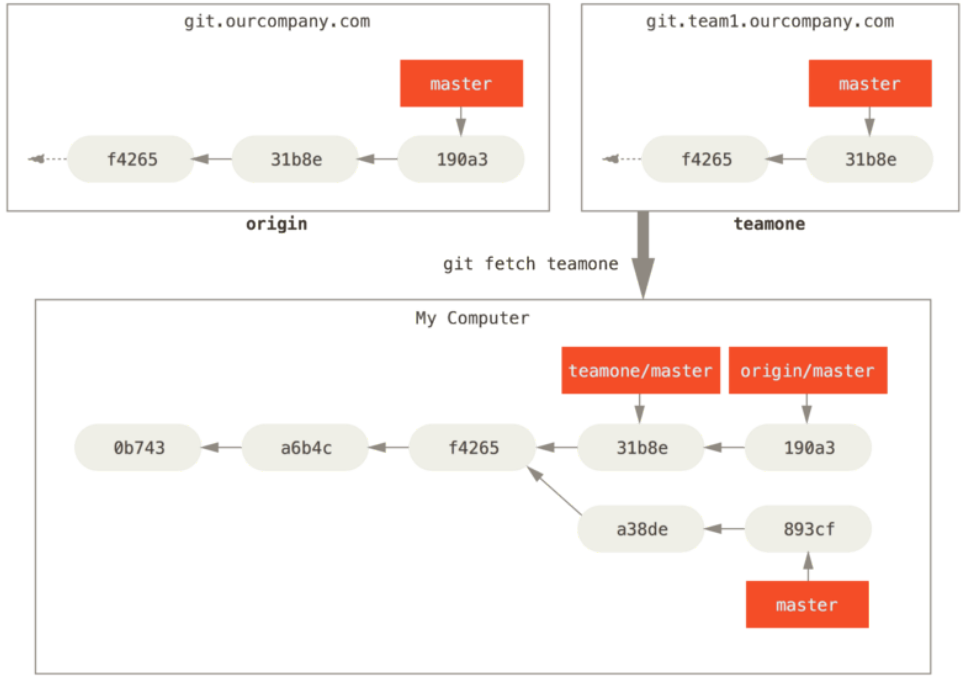
- You can have many remotes



git remote add teamone git://git.team1.ourcompany.com



- ... `git fetch` to synchronize their state to your repository



- You can `fetch/merge` or `pull` from remote-tracking branches without creating local branches
- If you want to work (and perhaps contribute) to a remote branch, you can create a local branch out of it:
    - `git checkout <branch>` (if no name conflict, will just create a tracking branch
    - `git checkout -b <branch> --track <remote/branch>` (specific, in case name there are name conflicts)

Let's see a demo

demo-1-6.cast (https://asciinema.org/a/wlBMwwJXF24xYPqzfi3QS91SG?autoplay=1&cols=180&rows=40) (tracking branches, and "local" remotes)

Practice time

- In *git-it-cp2* follow similar steps to the previous demo, so that:
    - you set the local repository *git-it-cp1* as a remote
    - checkout the branch test_branch1
    - commit some change to it
    - push it to the repository *git-it-cp1*
        - if you just do `git push` it should work, (do you understand why?)
        - when you are not in a tracking branch you can also push it to a remote branch, (do you understand how?)

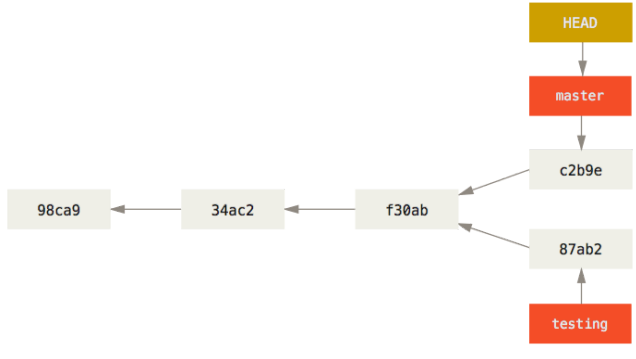
Most of the time, just one remote (probably GitHub)

- GitHub is just another remote, very useful for collaboration, but just another remote
- Creating remotes in other machines or even in different directories as we did can be very useful. I use it regularly:
    - one PC has required GUI software,
    - the other PC required documentation software.
    - I need to push changes quickly from one to the other without making many commits in the common repository

Now you understand branches (local and remotes) properly, but with several branches trouble will inevitably knock on your door at some point...

The problem: divergent branches (applies to local and remote branches)

- Before we created a *testing* branch and committed some changes.
- Later we went back to the *master* branch.
- At that point, if you commit to master, you end up with divergent branches.
- If you want to incorporate the changes in *testing* to *master*, you might have conflicts (maybe both commits updated the same function).



Two possible solutions to divergent branches

`git merge`

Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch.

`git rebase` (*see Section 7*)

Apply all changes made in the current branch on top of another branch. (*This is a more advanced command, and you have to be careful with it*)

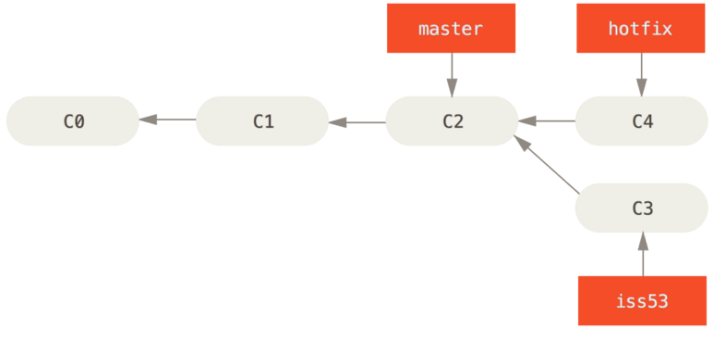
5. Merging

Merging

- Fast-forward merges
    - the simplest, we already saw an example of this in *demo-1-4.cast*
- No-conflict merges
    - very simple, and very common if working on your own
- Merges with conflicts
    - very usual when collaborating with others, specially if long time between commits

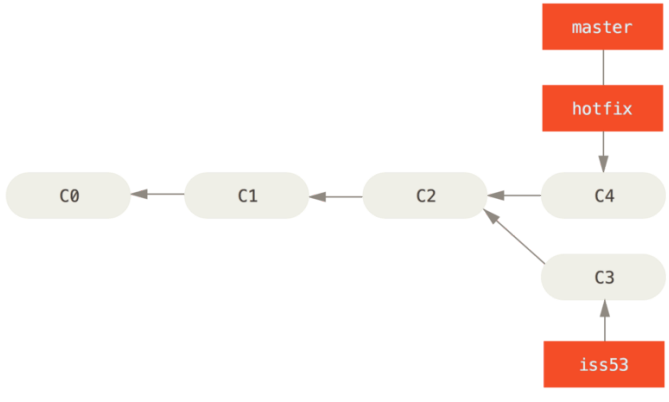
Fast-forward ”merge”

- Imagine you are in a situation like this, where:
  - you have two branches, started off the *master* branch, where you made one commit to each.
  - you want now to incorporate to *master* the changes done in branch *hotfix*



Fast-forward ”merge” (2)

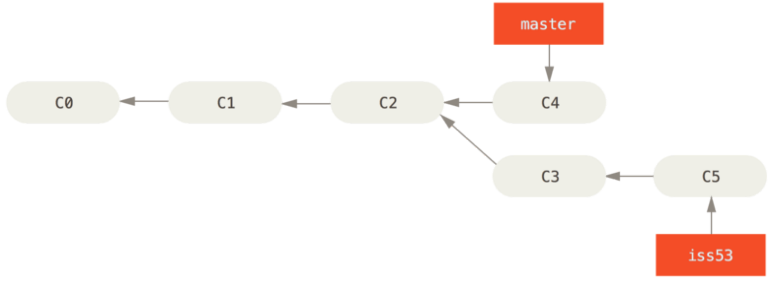
- Check out the master branch and merge the hotfix branch
    - This is a fast-forward merge (i.e. nothing really to merge, not divergent branch)



demo-1-7.cast (https://asciinema.org/a/ZbFhHkGvVaIGWF3HiF9ZiSyUG?autoplay=1&cols=180&rows=40) [Note the *behind*, *ahead* comments when doing `git branch -vv`]

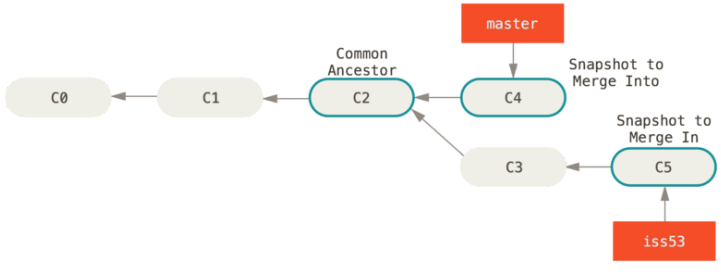
No-conflicts merge

- Following from the previous situation:
  - you now have a branch *iss53* that has diverged from the *master* branch
  - a "fast-forward" is not possible when incorporating those changes to *master*



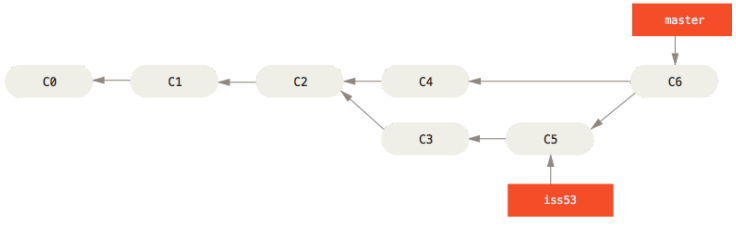
No-conflicts merge (2)

- A `git merge` command will have to work harder now to find:
  - which commit is common to both lines
  - what changes were made in each branch so as to collect all changes
  - if changes in, e.g. different files/functions, merge possible w.o. conflicts



No-conflicts merge (3)

- And we end up with a new "merge" commit in *master*:



Note: remember that `git merge iss53` means: merge branch iss53 to my current branch (i.e. you want to issue the merge command with the destination branch checked-out)

demo-1-8.cast (https://asciinema.org/a/4SIf3zhP3K0ds9I5SdmHXUe8d?autoplay=1&cols=180&rows=40)

Merge with conflicts

When git cannot merge automatically, it will tell you:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

The conflicting files will have conflict-resolution markers:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">contact us at support@github.com</div>
>>>>>>> iss53:index.html
```

Merge with conflicts (2)

- To resolve the conflict by hand:
    - just edit the file with your usual text editor, leaving the correct resolution (and no markers)
    - run `git commit` when all conflicts have been resolved

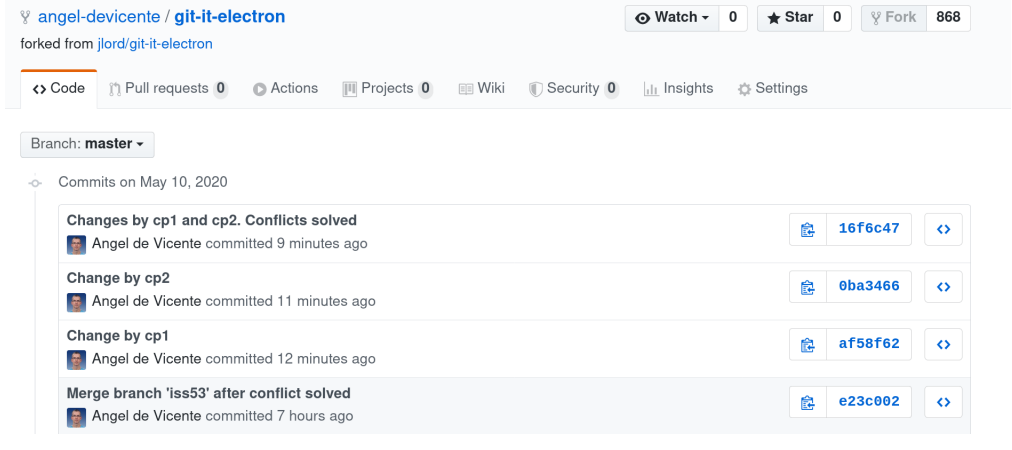
      demo-1-9.cast (https://asciinema.org/a/MxjPsupBk3toHCQsOuEQBKbMF?autoplay=1&cols=180&rows=40)
- But if collaborating regularly you should really learn how to use an external tool
    - `git mergetool` will tell you options and how to configure git
    - GitAhead demo: https://www.youtube.com/watch?v=W-FHwUwE84M (https://www.youtube.com/watch?v=W-FHwUwE84M) (many more GitAhead videos at https://www.youtube.com/playlist?list=PLkhgTa3ULplz_DgalwtORMviEJeHy07EB (https://www.youtube.com/playlist?list=PLkhgTa3ULplz_DgalwtORMviEJeHy07EB))
    - Emacs + Magit + Ediff demo: https://youtu.be/S86xsx_NzHc (https://youtu.be/S86xsx_NzHc)

Practice time

- First make sure you synchrnonize the *master* branch in *git-it-cp1* and *git-it-cp2* with GitHub
    - If you followed the examples above,
        - *git-it-cp1* is synchornized with *origin/master*
        - *git-it-cp2* has un-pushed commits. Push those ones to GitHub
        - then pull them from *git-it-cp1*
    - In any case, make sure that *master* in *git-it-cp1*, *git-it-cp2*, and *origin/master* are in the same state.
- Make a commit (adding one line to a file) in *git-it-cp1* and push to GitHub
- In *git-it-cp2* make a commit (adding one line to the same file above) and push to GitHub
    - You won't be able to do it, since now the remote branch is ahead, so you will have to do a *fetch*
    - Then *merge origin/master*
        - And now you should get conflicts. Solve the conflicts and push to GitHub
- Back in *git-it-cp1*, pull (or fetch/merge) the changes that you just did in *git-it-cp2*

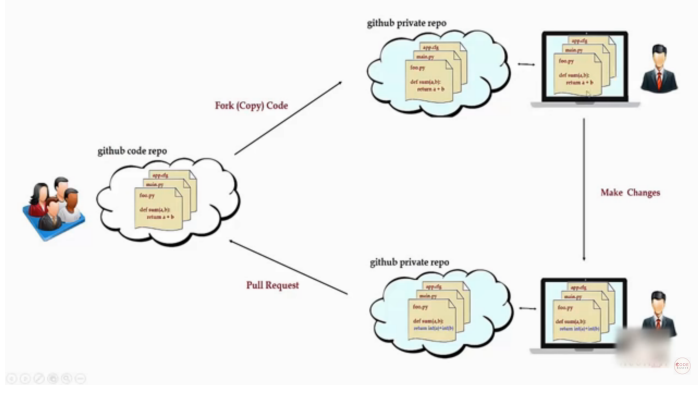
Possible solution

[*demo-1-10.cast*](https://asciinema.org/a/1tHAh1ygCDswdlTaG64KPuk91?autoplay=1&cols=180&rows=40)



Pull/merge requests (GitHub)

- Very useful when contributing code when not a collaborator
- (but it can also be used across branches with collaborators)



From https://www.youtube.com/watch?v=e3bjQX9jIBk (https://www.youtube.com/watch?v=e3bjQX9jIBk)

Pull request GitHub live demo

- Pull request all inside GitHub
- Pull request via local branches for more advanced control
- Keeping your fork-ed version up-to-date with the original (*upstream*)

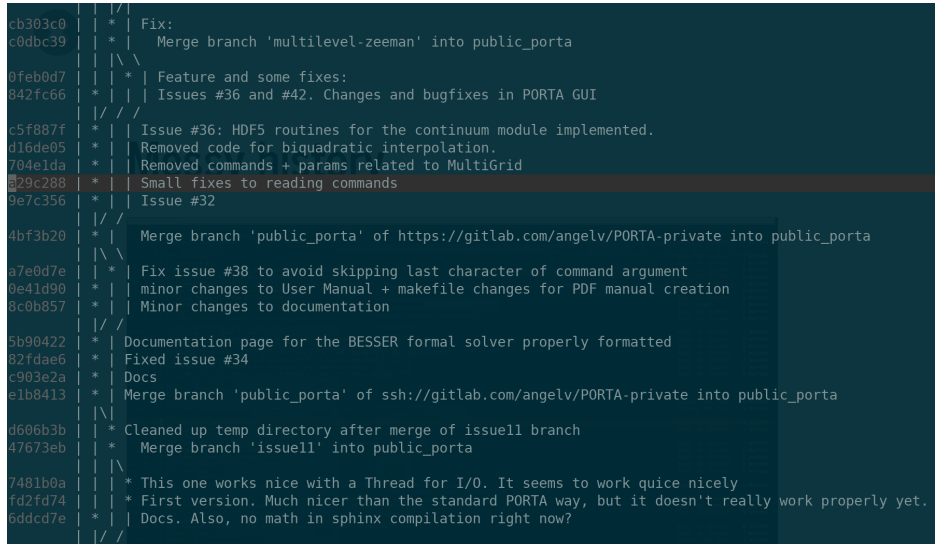
6. Workflows

Workflows

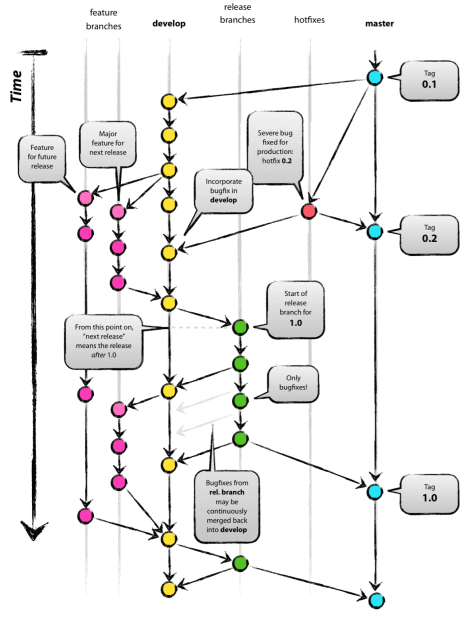
- Branches are a *killer* Git feature and you should use them often
    - create a branch to fix bugs
    - create a branch to test new ideas
    - etc.
- But you need to get organized with workflows or things will get messy quickly. Useful commands:
    - `git branch --merged`
    - `git branch --no-merged`
    - `git branch -d <branch>`

Messy history

- Our own project, only two/three active developers

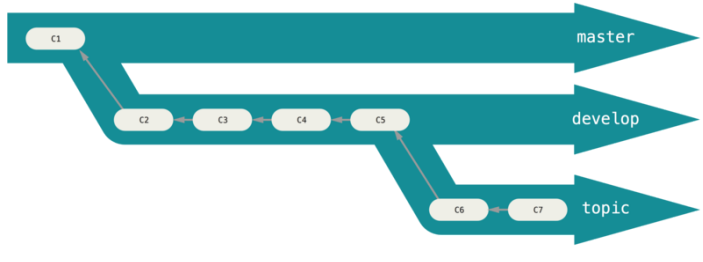


- Example workflow ([https://nvie.com/posts/a-successful-git-branching-model/ (https://nvie.com/posts/a-successful-git-branching-model/)](https://nvie.com/posts/a-successful-git-branching-model/))

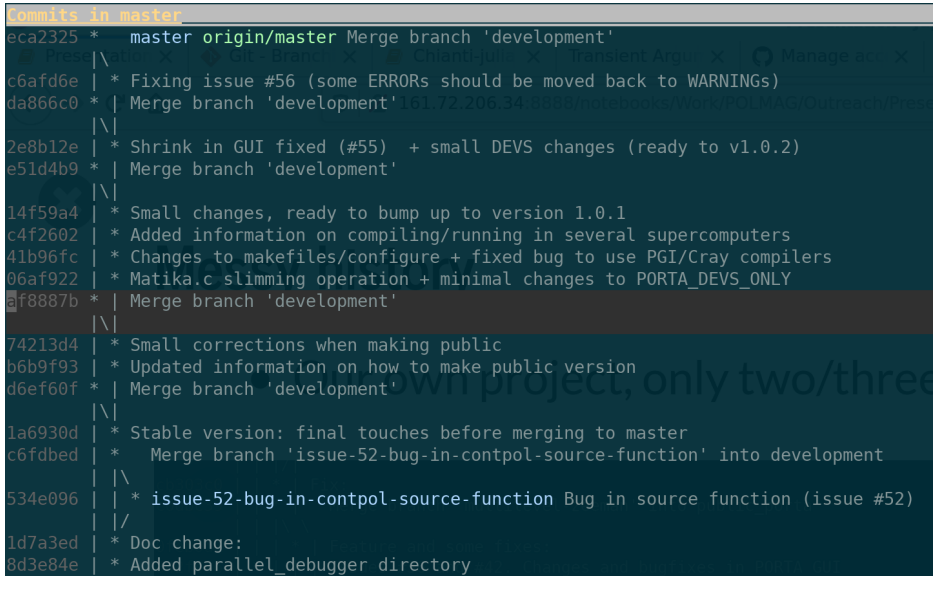


Many types of workflows

- see, for example, https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows (https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows)
- for our development of PORTA (https://gitlab.com/polmag/PORTA (https://gitlab.com/polmag/PORTA)), we decided to have something similar to:



Clean history



Our workflow for PORTA

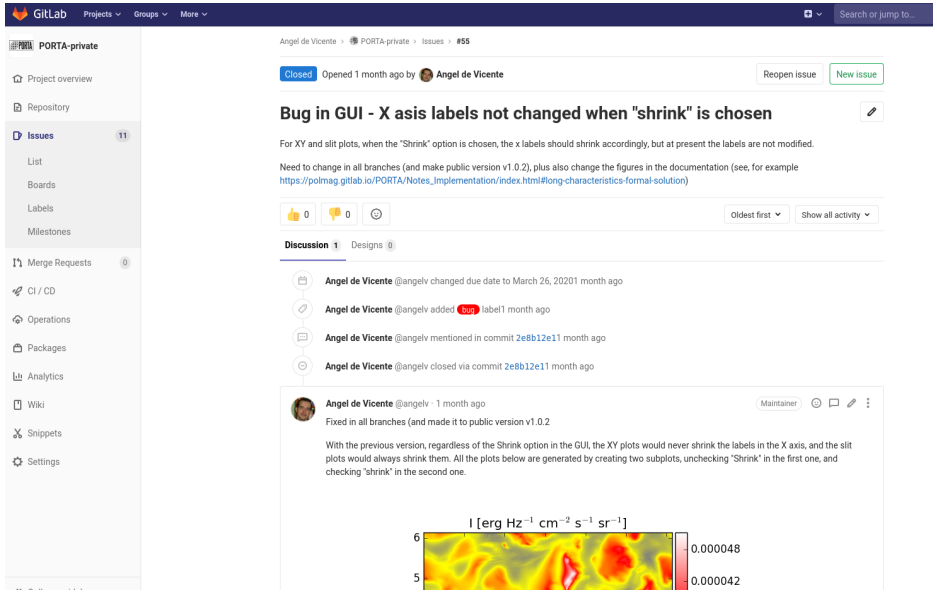
- Three long-lived branches: *release*, *master*, *development*
- Short-lived topic branches for fixes, bugs, new developments, etc.
- After merged to *development* we delete the topic branch.
- For very small issues, we commit directly to *development*.
- For larger issues, we create a topic branch, and follow it up via an *issue* (in GitLab).
- Developers in the team have direct access only to *development*

- The repository maintainer (i.e. me) takes care of merging into *master* (stable versions) and *release* (ready to become public versions)

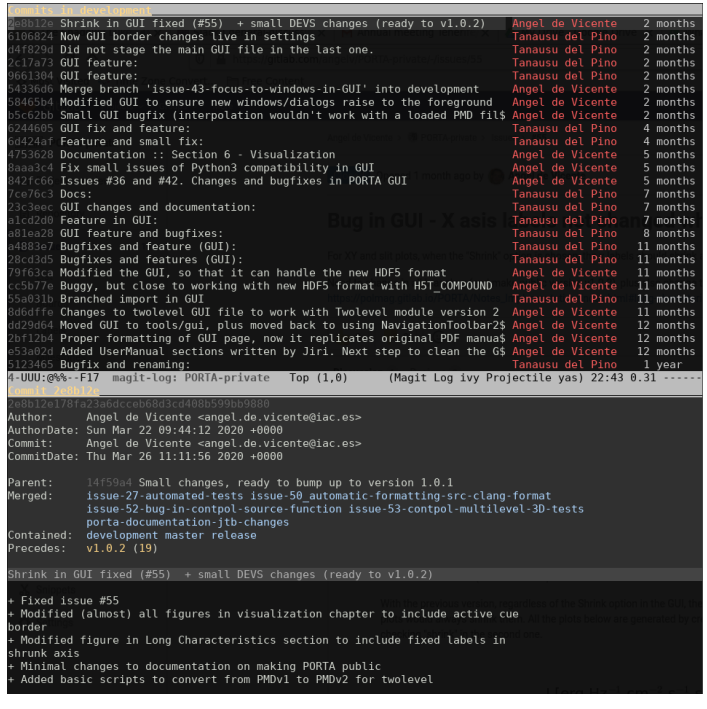
- To keep a clean history, we make sure that topic branches (**for this, we need to make use of rebasing**):

    - can be merged as fast-forward
    - but we merge it as no fast-forward to keep a more readable history

Issues in GitLab (very similar in GitHub)



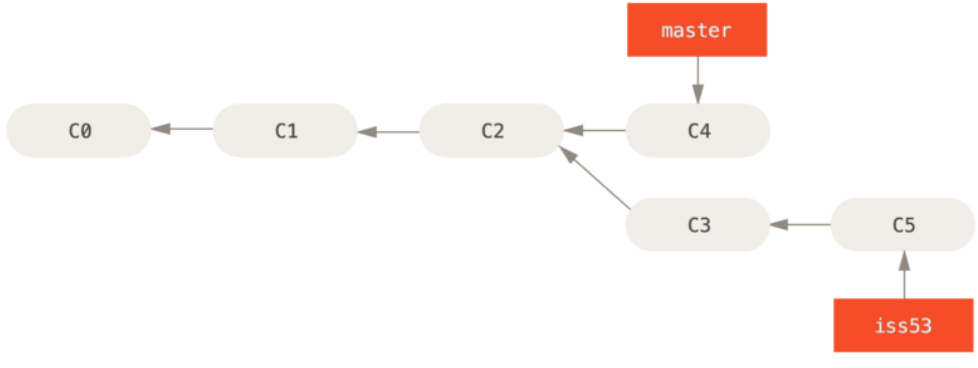
Relate issues with logs



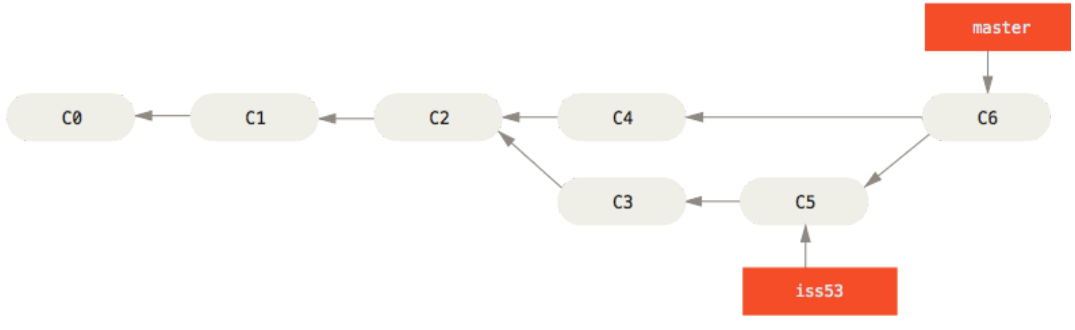
7. Rebasing

Why do we want to avoid regular merges?

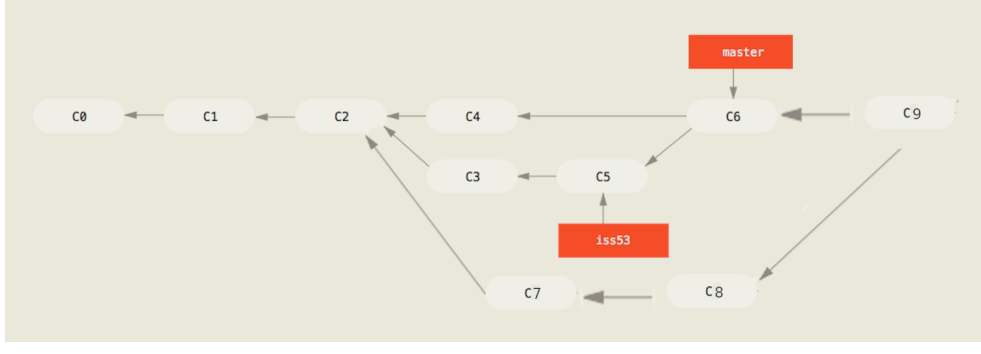
- Remember how we do merges of divergent branches



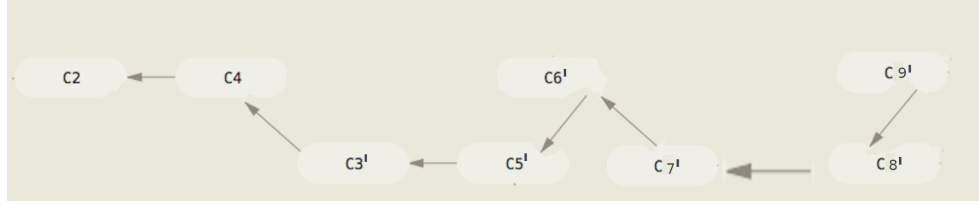
- Regular merge
    - this could be two developers, each working in a branch



- Regular merge (2)
  - ... things start to get messy if several branches/developers want to merge

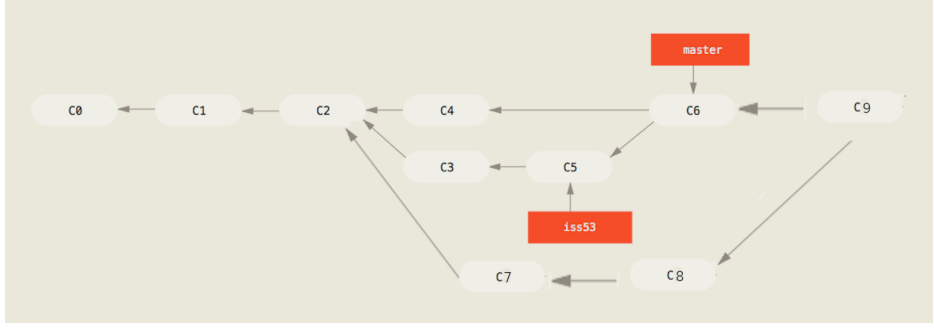


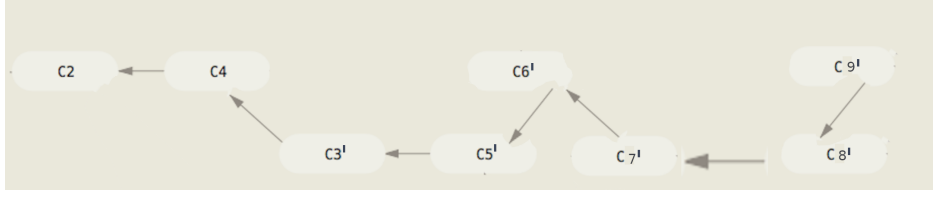
- Rebasing
    - Replay the commits in this branch, but starting from some other branch



Merge vs. rebase

Let's just see them side by side, to understand the differences





Rebasing to keep clean commits

- While rebasing you can decide that some commits can be *squashed*, "skipped*, put in different order, etc. If you do this only in local branches, it is very useful, for example, to combine half-baked commits into a coherent one.
- But you need to have one rule in mind: **never rebase public branches** (if other people fetched your history and you rebase it, you are rewriting it, which will cause a lot of trouble for them)
- More info about rebasing: https://git-scm.com/book/en/v2/Git-Branching-Rebasing (https://git-scm.com/book/en/v2/Git-Branching-Rebasing)

8. Other useful Git configuration/commands/tools

- .gitignore (patterns of files that Git will simply ignore)
- reset (discard commits in a private branch or throw away uncommited changes)
- cherry-pick (Apply the changes introduced by some existing commits)
- bisect (Use binary search to find the commit that introduced a bug)
- blame (Show what revision and author last modified each line of a file)
- submodules