Git basics

Ángel de Vicente (*angel.de.vicente@iac.es*)

Date: November 10, 2020

Index

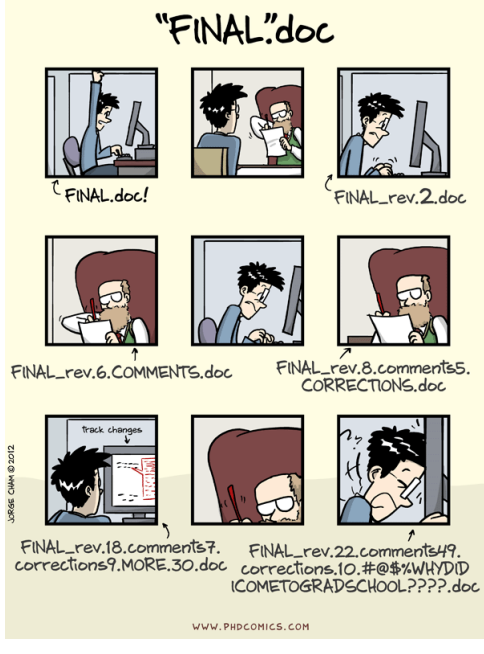
1. Git basics

Basic concepts:

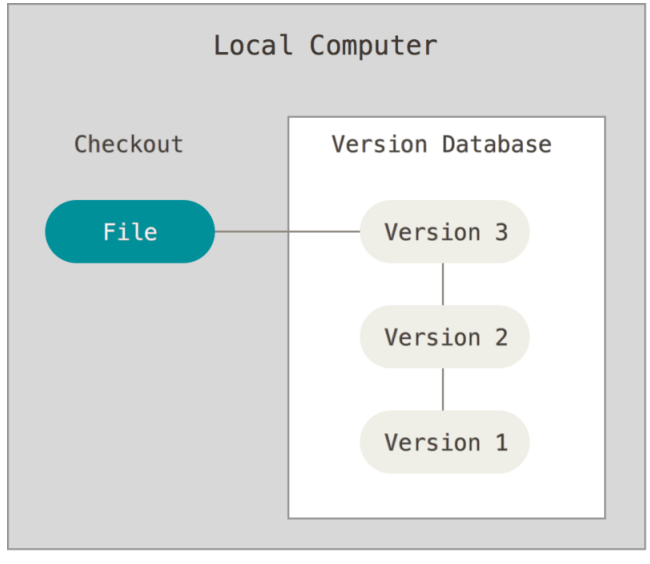
- What is version control
- What do we version control (text vs. binary files)
- What is a version control system
- You have probably already used a version control system directly or indirectly (e.g. Dropbox, Overleaf, etc.)
- Version control ideas: record/playback, branching, merging
- Benefits of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

Types of Version Control Systems

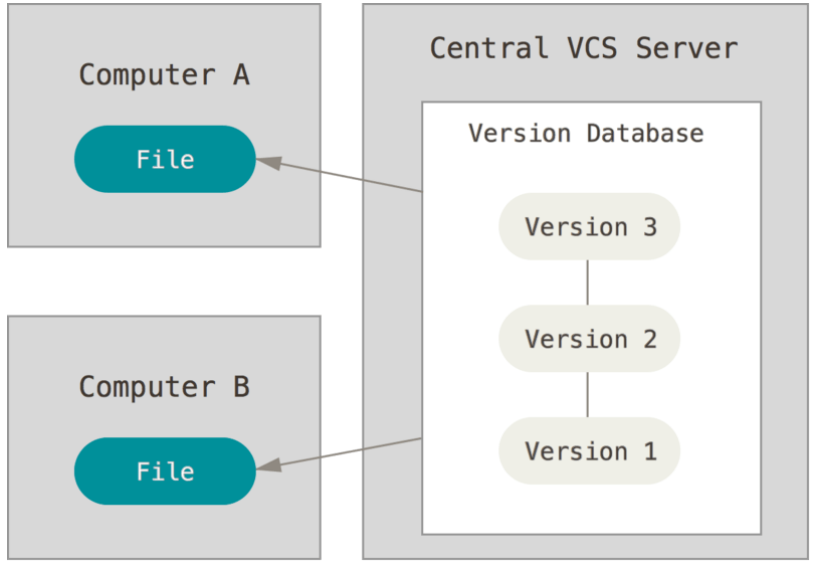
- No VCS (*don't be the one doing this!*)



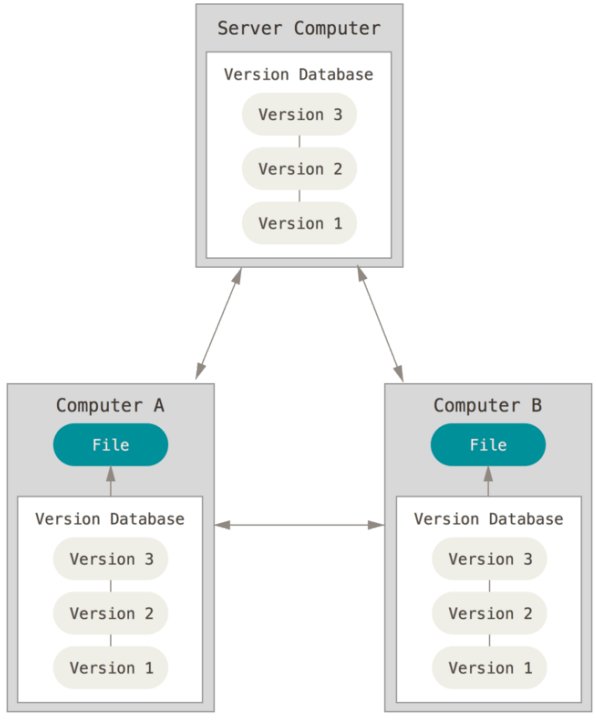
- Local VCS (e.g. RCS)



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



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



Setting up git

In [3]:
```
! git --version
```

```
git version 2.28.0
```

- *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 [7]:
```
! 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
pull.rebase=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
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

Getting help

In [19]:
```
! git --help | head -n 18
```

```
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
```

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
```

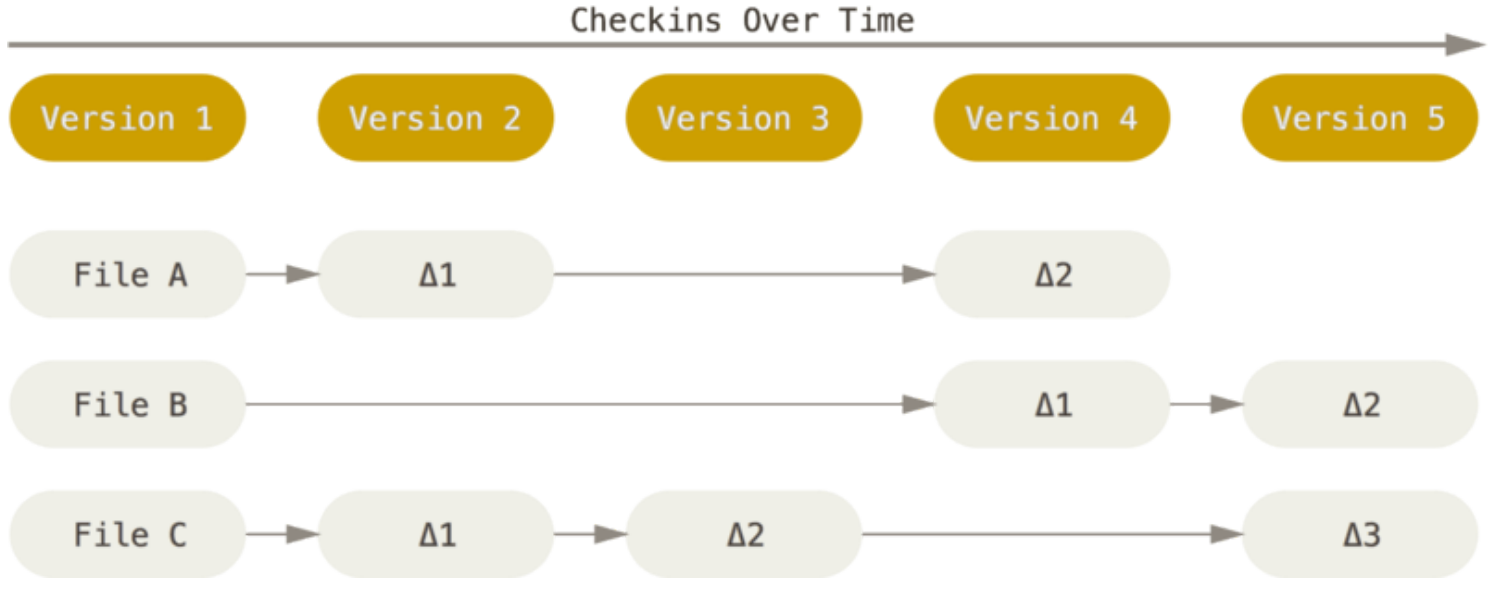
In [19]:
```
! git status --help | head -n 22
```

```
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.
```

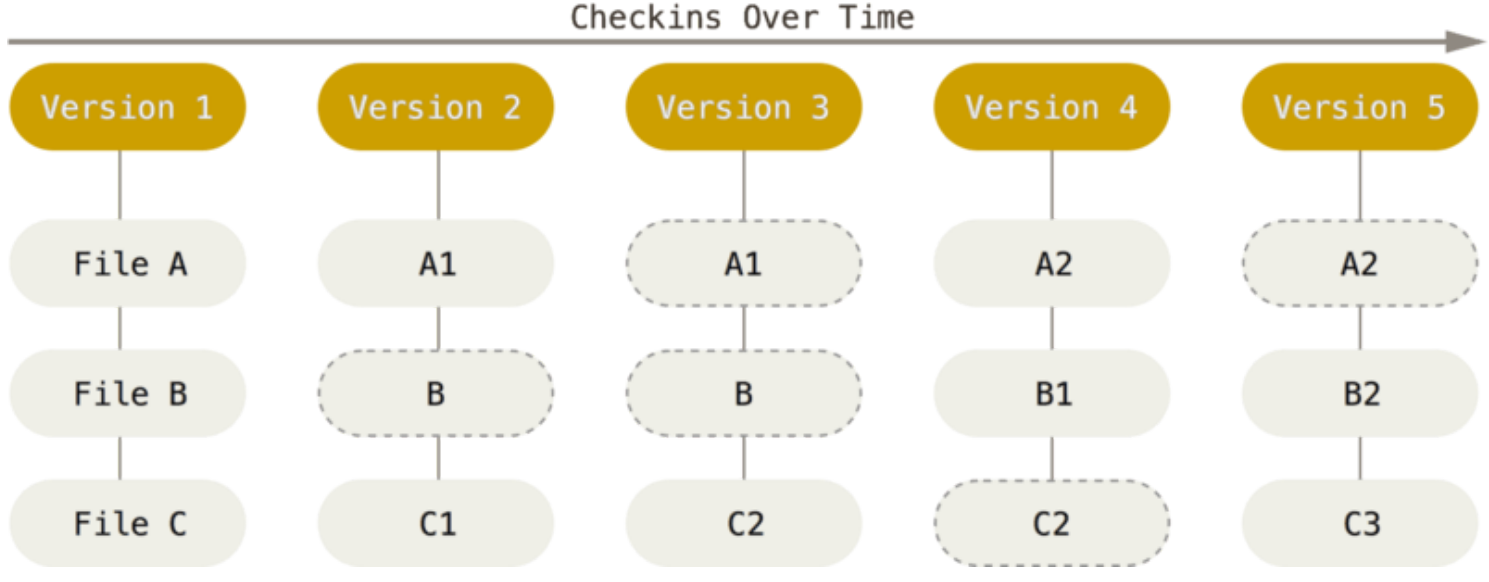
Main ideas in Git

- 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

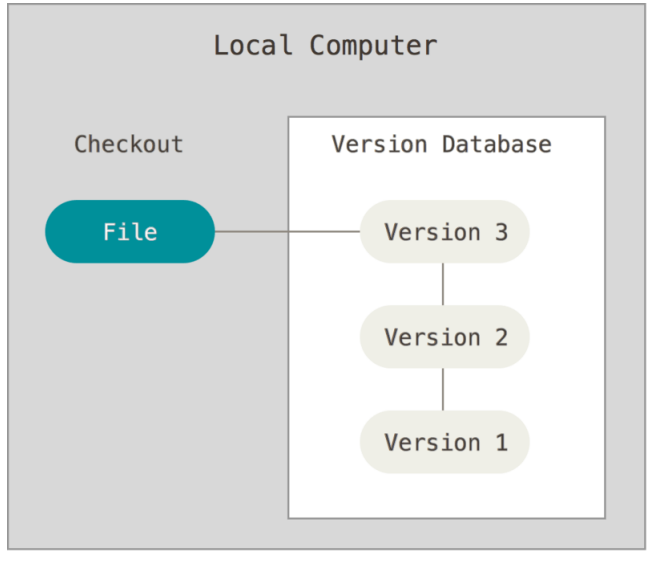


- Snapshots (Git)



2. Git basic usage

Using git as a local VCS



Create a repository

git init

In [15]:
```
%%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 Oct 28 11:43 .
drwxr-xr-x 6 angelv angelv 4096 Oct 28 11:43 ..
drwxr-xr-x 7 angelv angelv 4096 Oct 28 11:43 .git
```

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

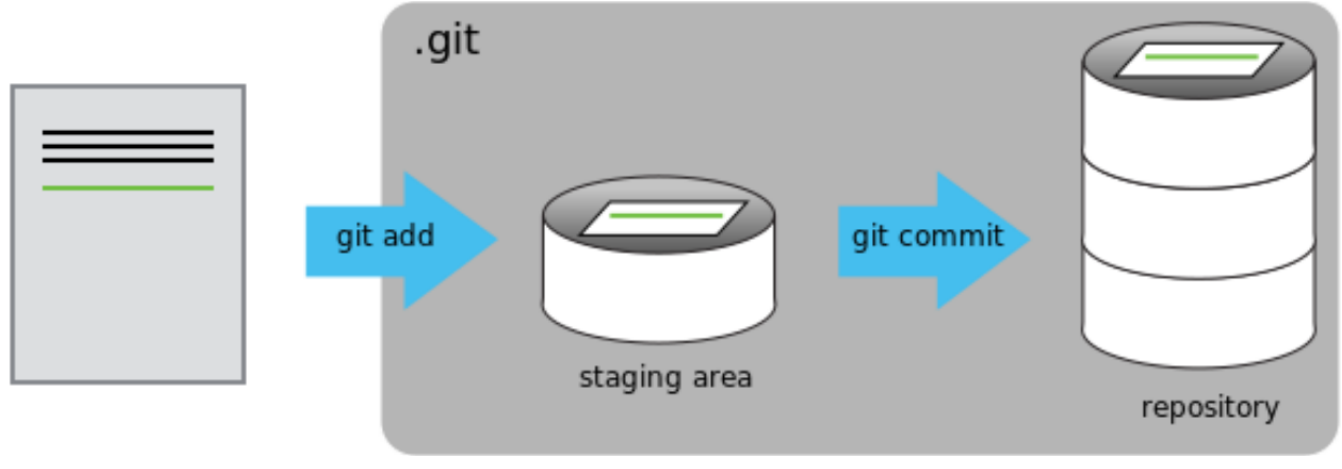
In [16]:
```
%%bash

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

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

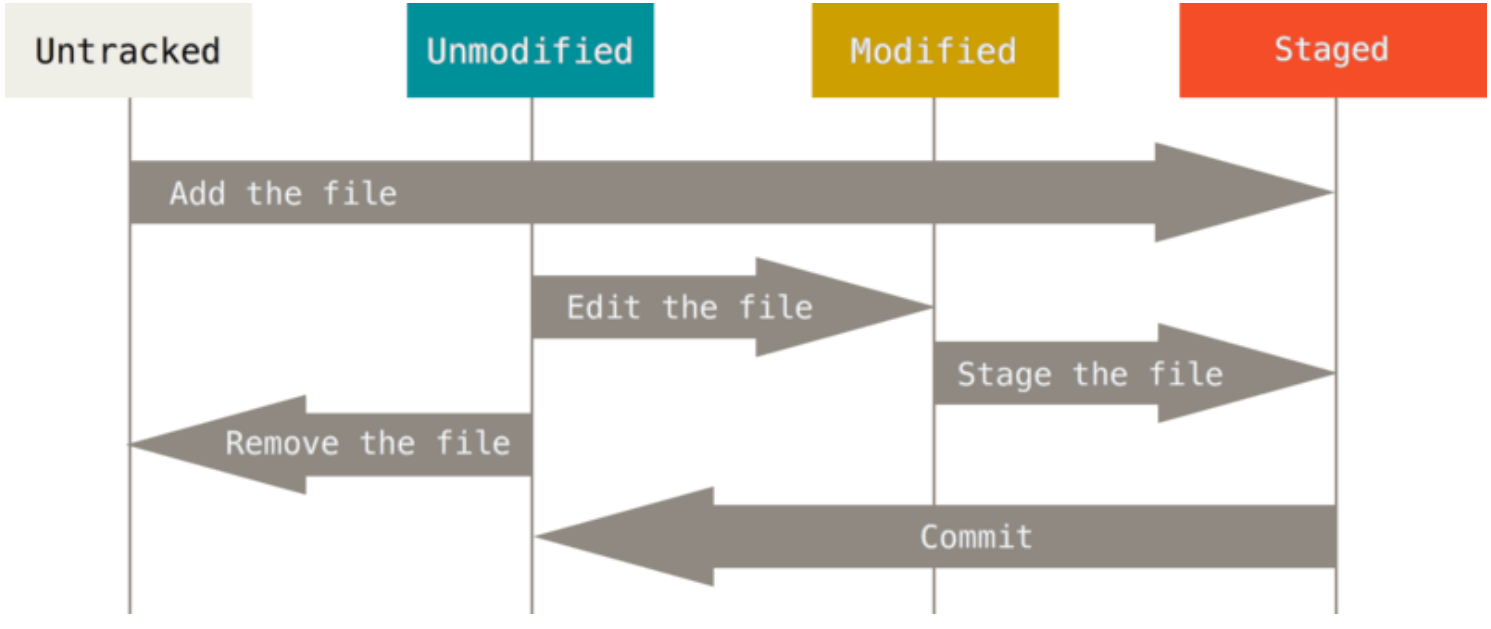
Recording changes

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

- Commiting changes



- Tracking changes



Recording changes demo

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

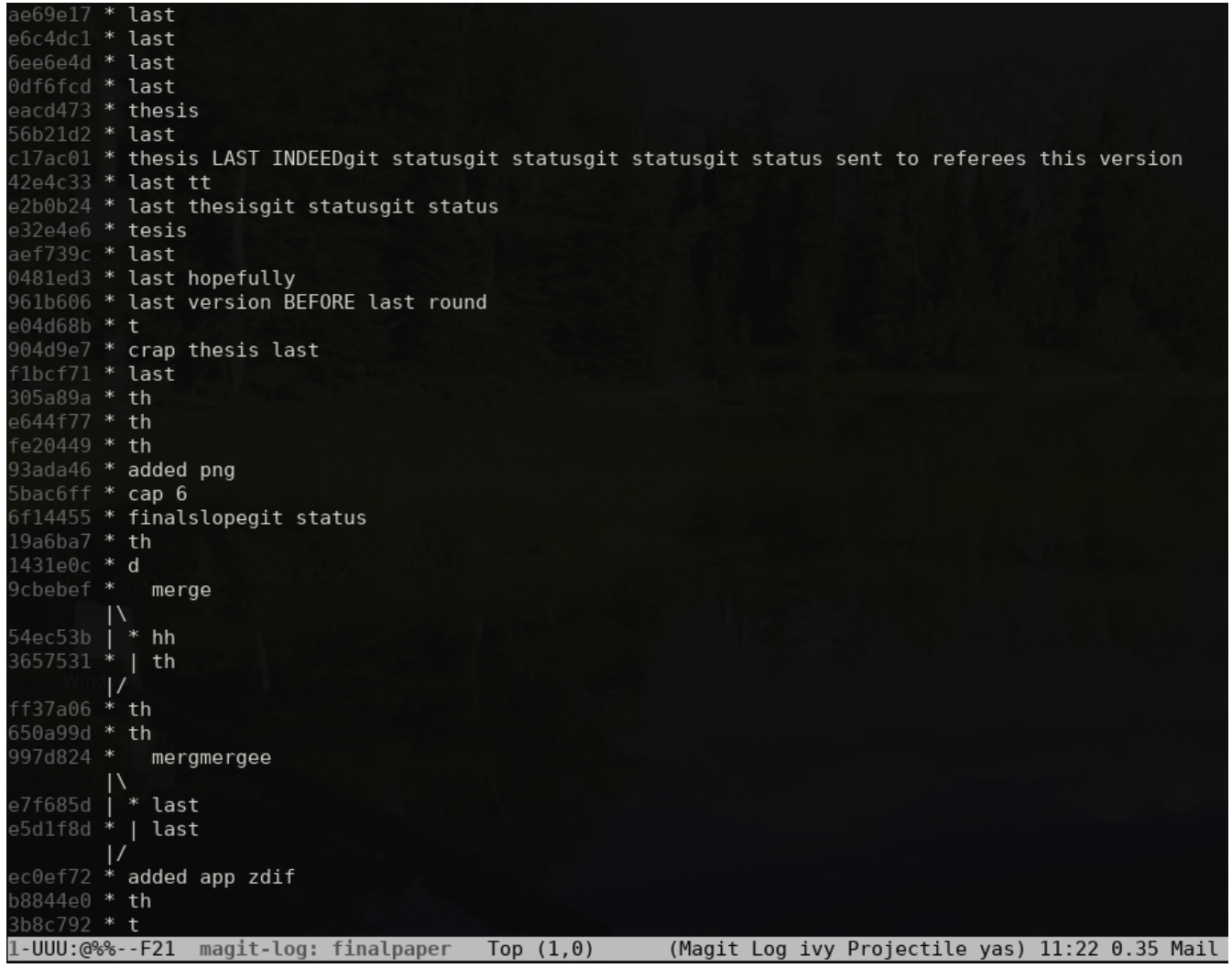
Viewing history

- `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)

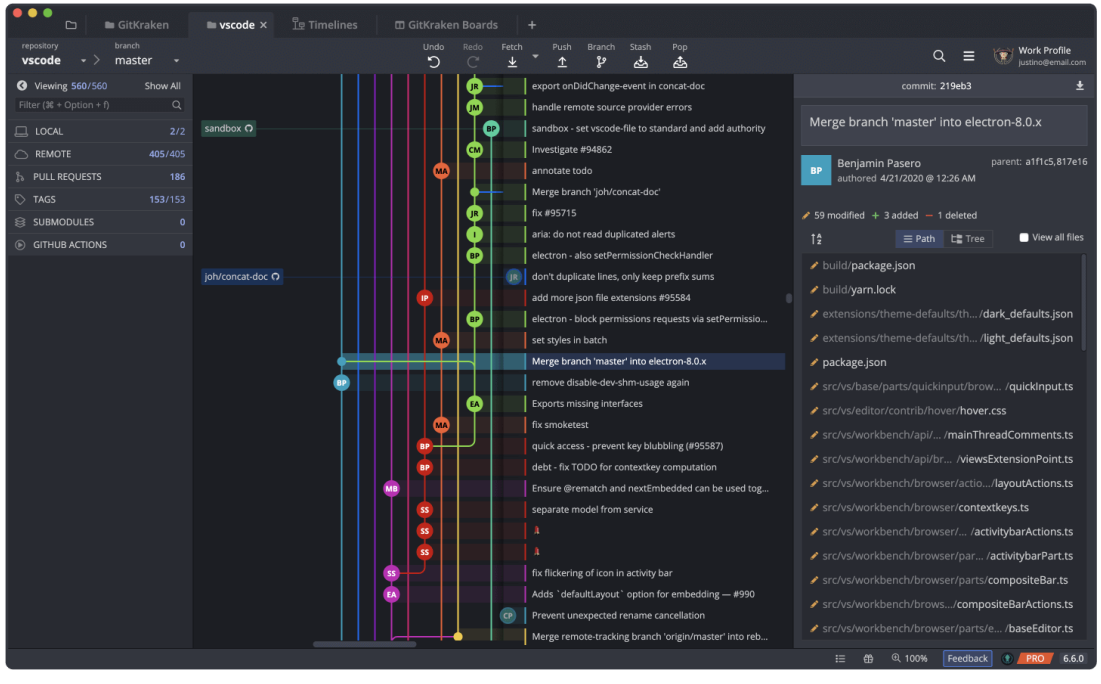
- In older versions of `git`, instead of `git restore` you would use:
  - `git reset HEAD ...`
  - and `git checkout ...`

    (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))

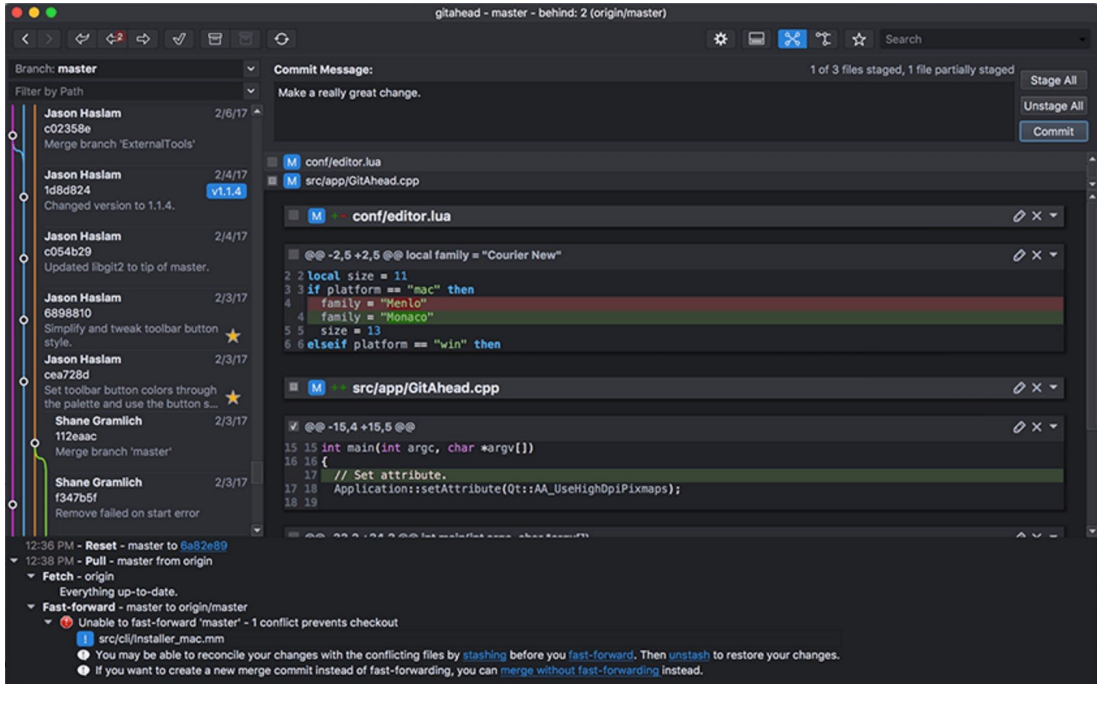
Command-line vs GUIs

- 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)

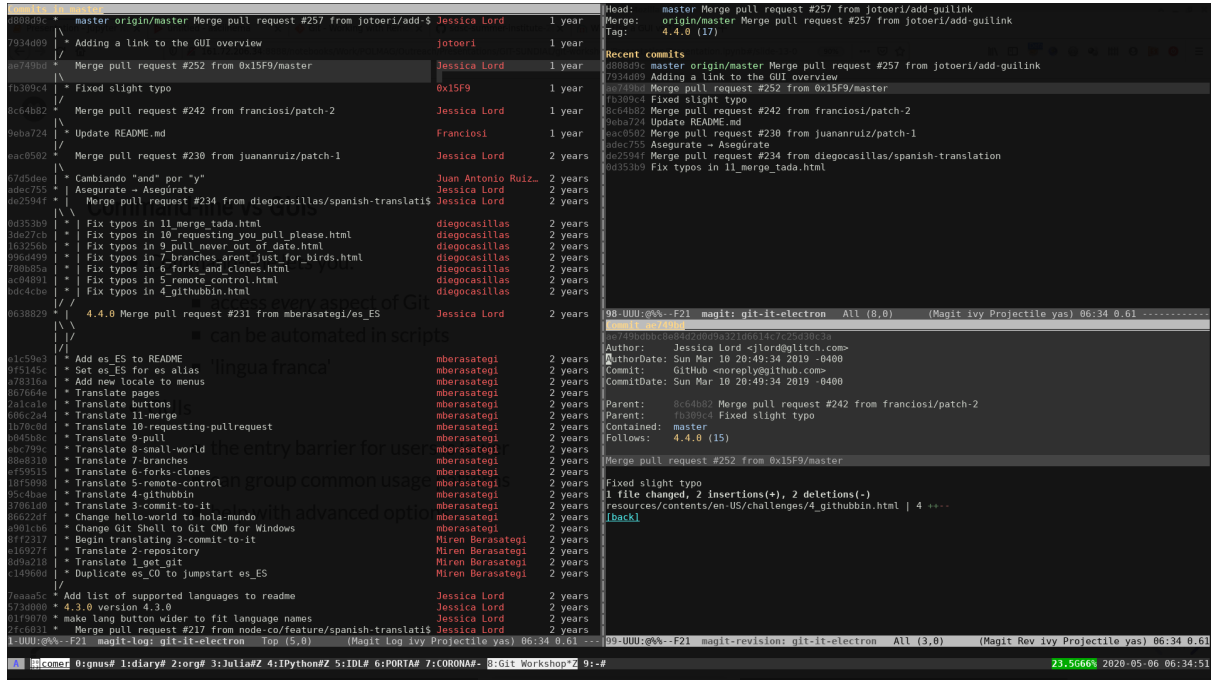
- GitKraken https://www.gitkraken.com/ (https://www.gitkraken.com/)



- 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 mini-demo with Magit

- See usefulness of hunks (peek into "Intermediate Git")

3. Branches

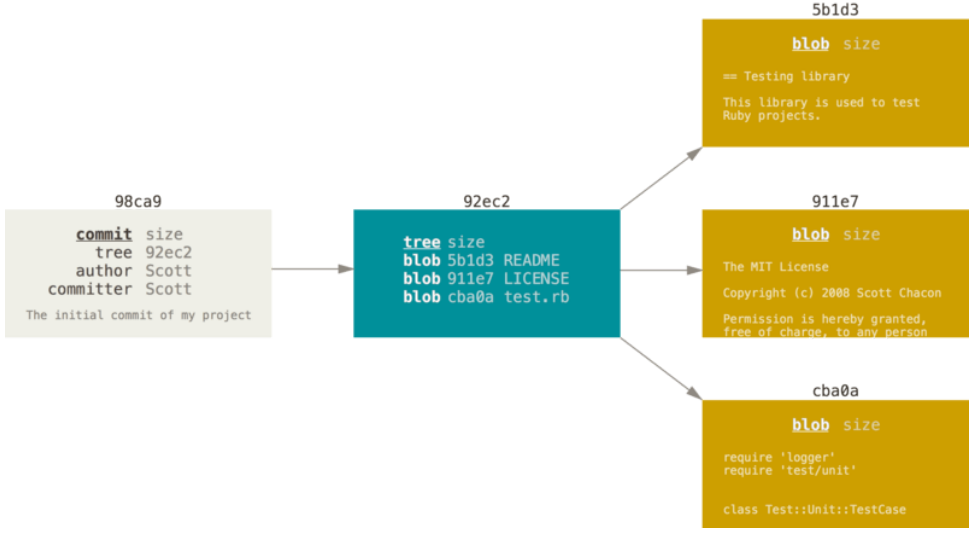
Branches

- 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.

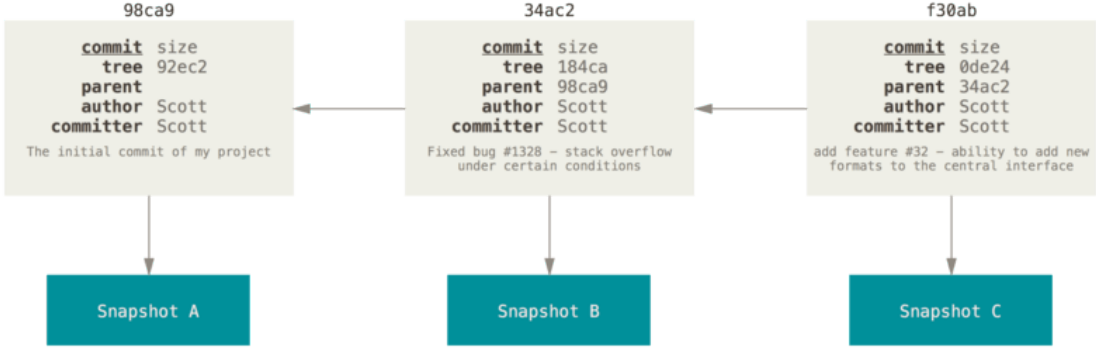
So, let's try and see some of the Git internals to understand what branches are...

(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))

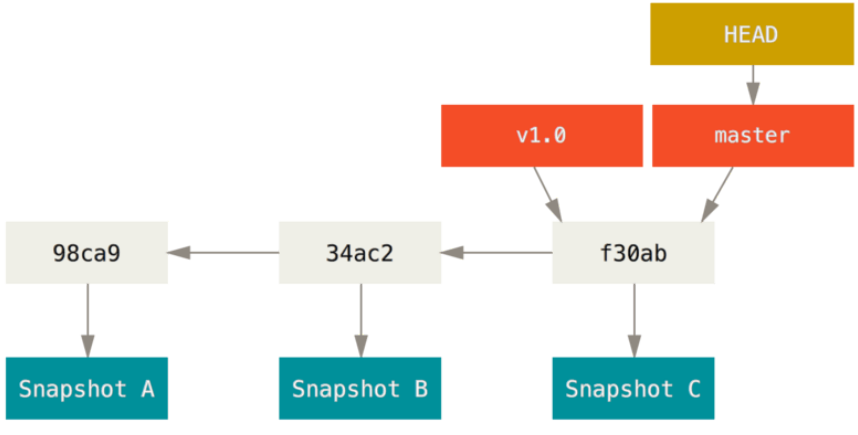
- A *commit* points to a *tree* object, and this to *blob*s
    - For us the important part is just the *commit*, which points to a *snapshot* of our work



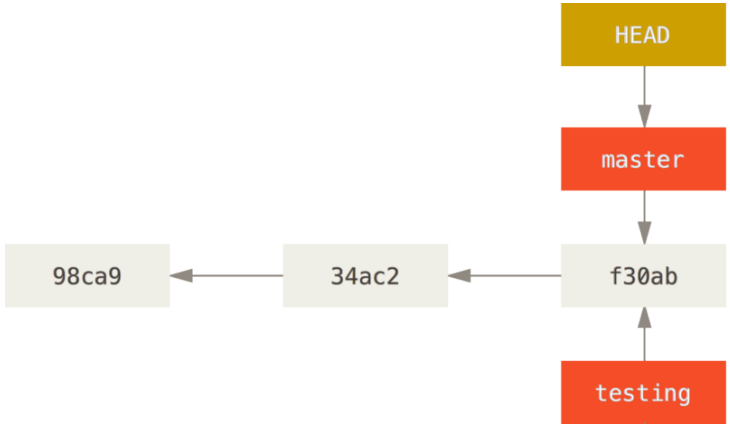
- Creating more *commit*s, keeps a linked list
    - (which is basically what you see when you do `git log`)



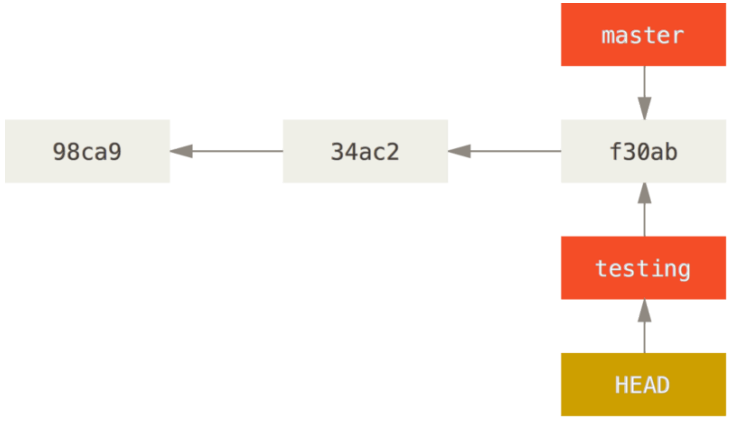
- A *branch* is just a pointer to one of these *commits*
  - *master* is created when you do `git init`, but it is no special
  - *HEAD* points to the branch we are working on



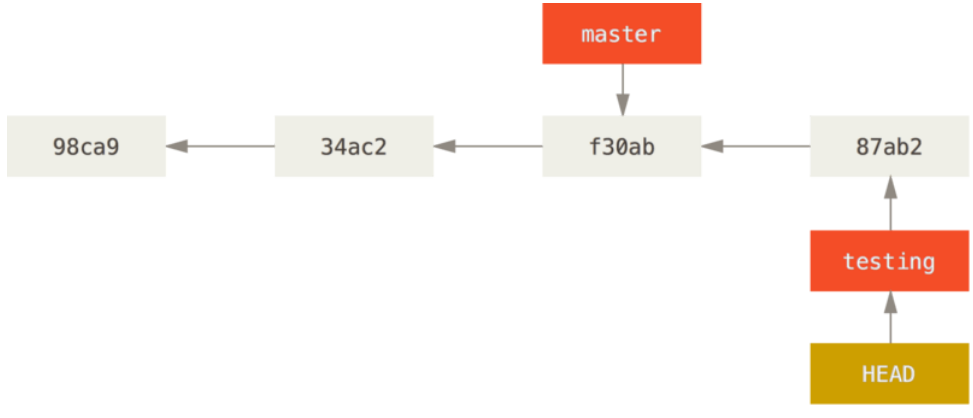
- Creating a branch
  - If we are working in *master* and we do `git branch testing`
    - *testing* branch is created, but we are still working on *master*
    - so you can see *HEAD* is still pointing to *master*



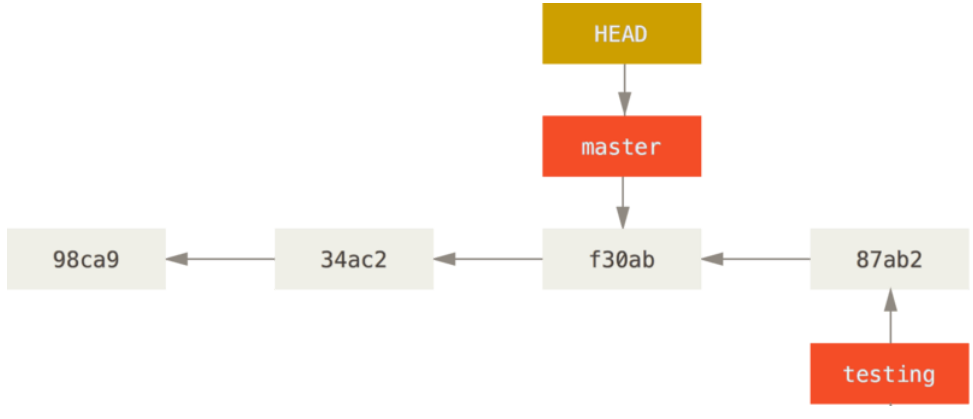
- To work with another branch
  - `git checkout testing` makes *HEAD* point to testing
    - when you switch branches in Git:
      - **files in your working directory will change.**
      - **if Git cannot do it cleanly, it will not let you switch at all.**



- 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)



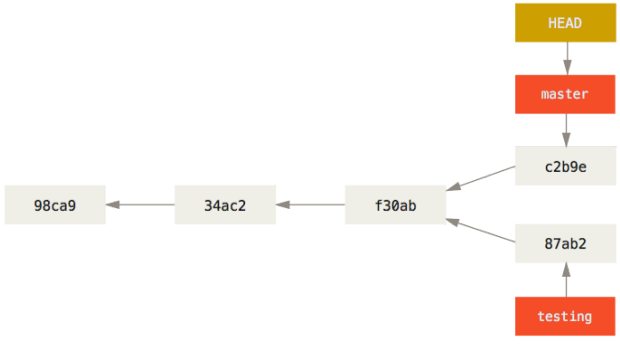
Branching demo

- 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)

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

- Earlier 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` (*look for "Intermediate/Advanced Git"*)

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*)

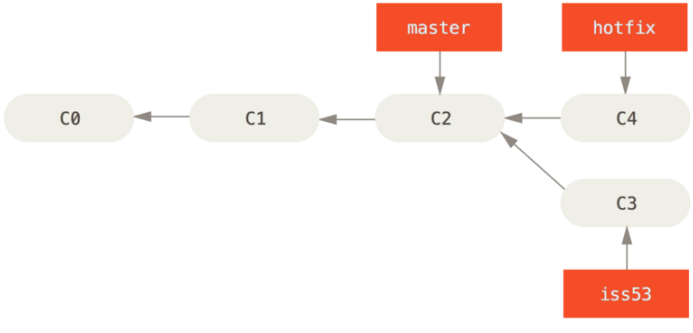
5. Merging

Merging

- Fast-forward merges
    - the simplest
- 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

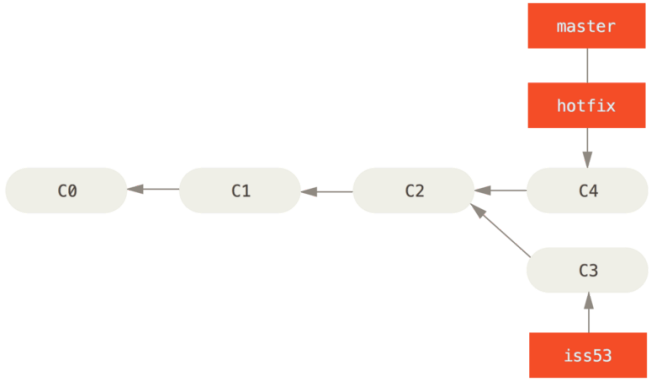
Fast-forward "merge"

- 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)

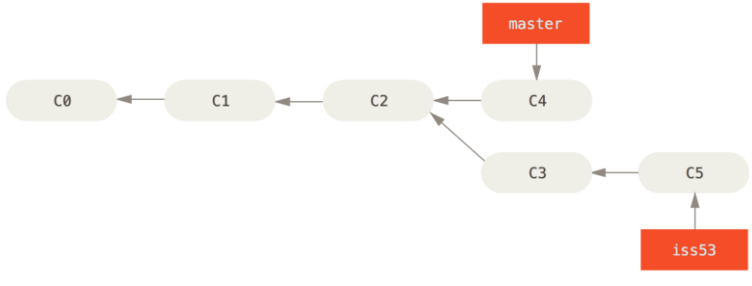
- Check out the master branch and merge the hotfix branch
  - This is a fast-forward merge (i.e. nothing really to merge, not divergent branch)



demo-1-7.cast (https://asciinema.org/a/ZbFhHkGvVaIGWF3HiF9ZiSyUG?autoplay=1&cols=180&rows=40)

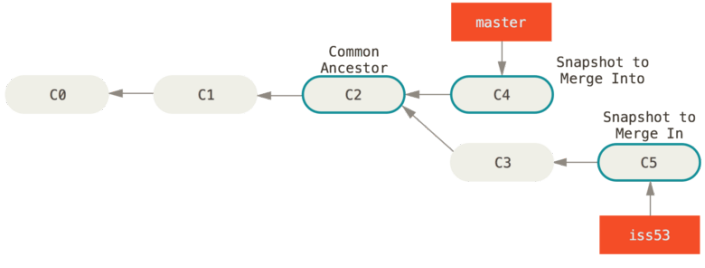
No-conflicts merge

- 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*



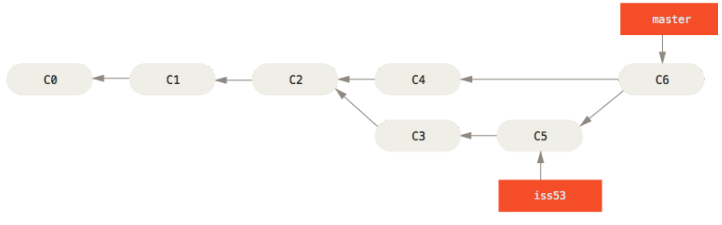
No-conflicts merge (2)

- 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 you should really learn how to use an external tool (peek into "Intermediate Git")
    - `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)
    - Emacs + Magit + Ediff demo: https://youtu.be/S86xsx_NzHc (https://youtu.be/S86xsx_NzHc)

References

- Git main page (https://git-scm.com/)
- Pro Git book (https://git-scm.com/book/en/v2)
- Git cheatsheet (https://ndpsoftware.com/git-cheatsheet.html)