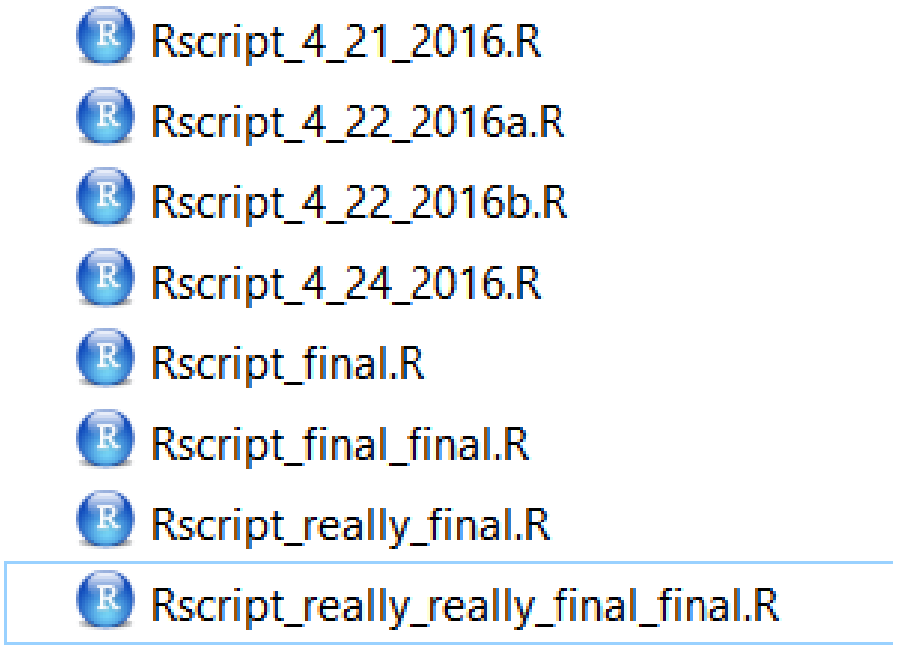
Git for Your Grandma and Other Parties

Sarah Habib

January 23, 2023

What Git?

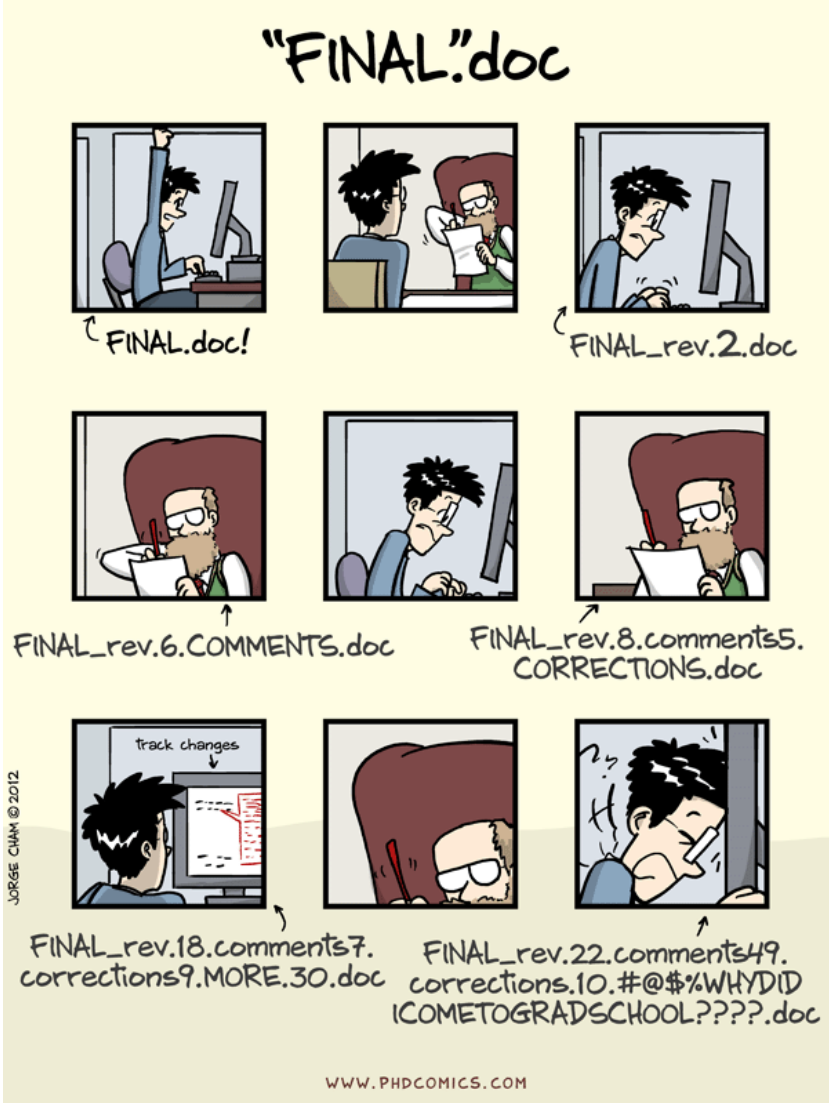
Version control is for tracking project changes.



Don't do this.

Why Git?

- It's space-efficient.
- It's safe.
- It gives a legible timeline.
- Experiment easily using branches.
- Collaboration is much simpler.
- Lightweight backups.
- Not just for code!



Architecture - Commits

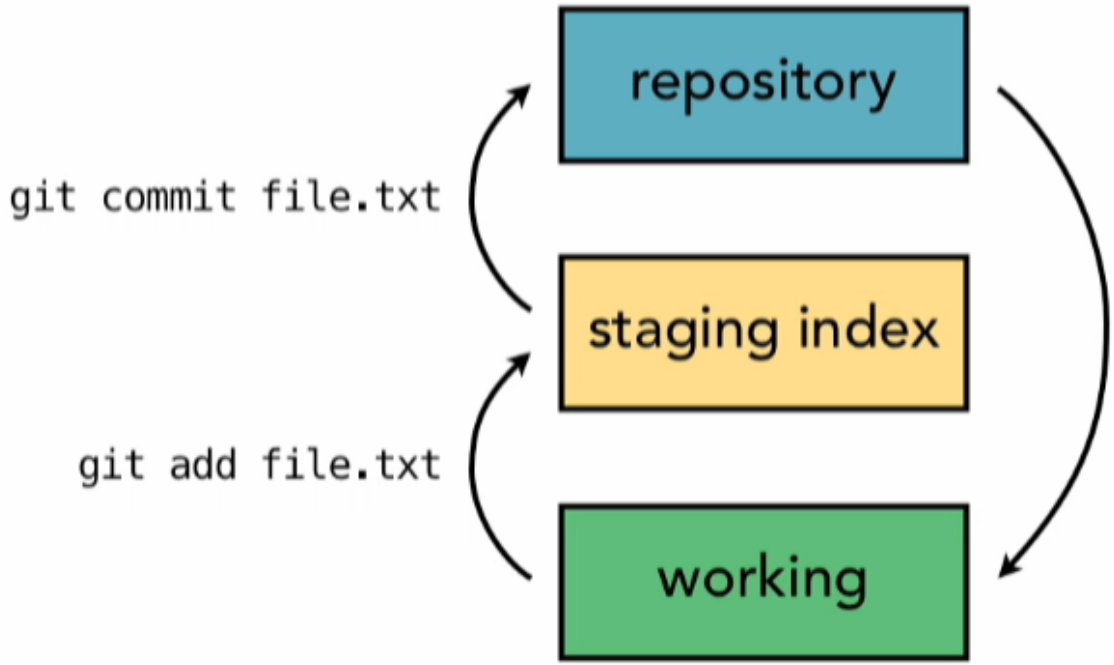
Git tracks project history as a collection of **commits**.

Commits are a snapshot of the project state containing:

- current versions of tracked files
- pointers to commits immediately before it (*parent* commits)
- a commit message (if you supplied one – which you better!)
- an identifying hash
- author name and email
- date and time of commit

Architecture - Three States

Git uses a "three tree" model to track project history. Files tracked by Git reside in at least one of three states:



- the **Working Tree** - modified
- the **Staging Index** - staged
- the **Commit History** - committed

Basic Functionality

`git init`

Create a new Git repo *with the current directory*. Works even in a directory that already contains files.

`git add <filenames>`

Add file changes to the staging index.

`git commit -m "<message>"`

Create a new commit from the currently staged changes.
`--amend` : Instead of making a new commit, add staged changes to last commit.

Basic Functionality - Other Useful Commands

`git status`

Gives a summary of files that have been modified and/or staged.

`git diff`

View unstaged file changes line-by-line.

`git log`

View a list of commits.

`git stash`

Save the current state of the working tree to a stack and reset the working tree to the latest commit.

`pop` : Restore the working tree state recorded at the top of the stack.

Branches

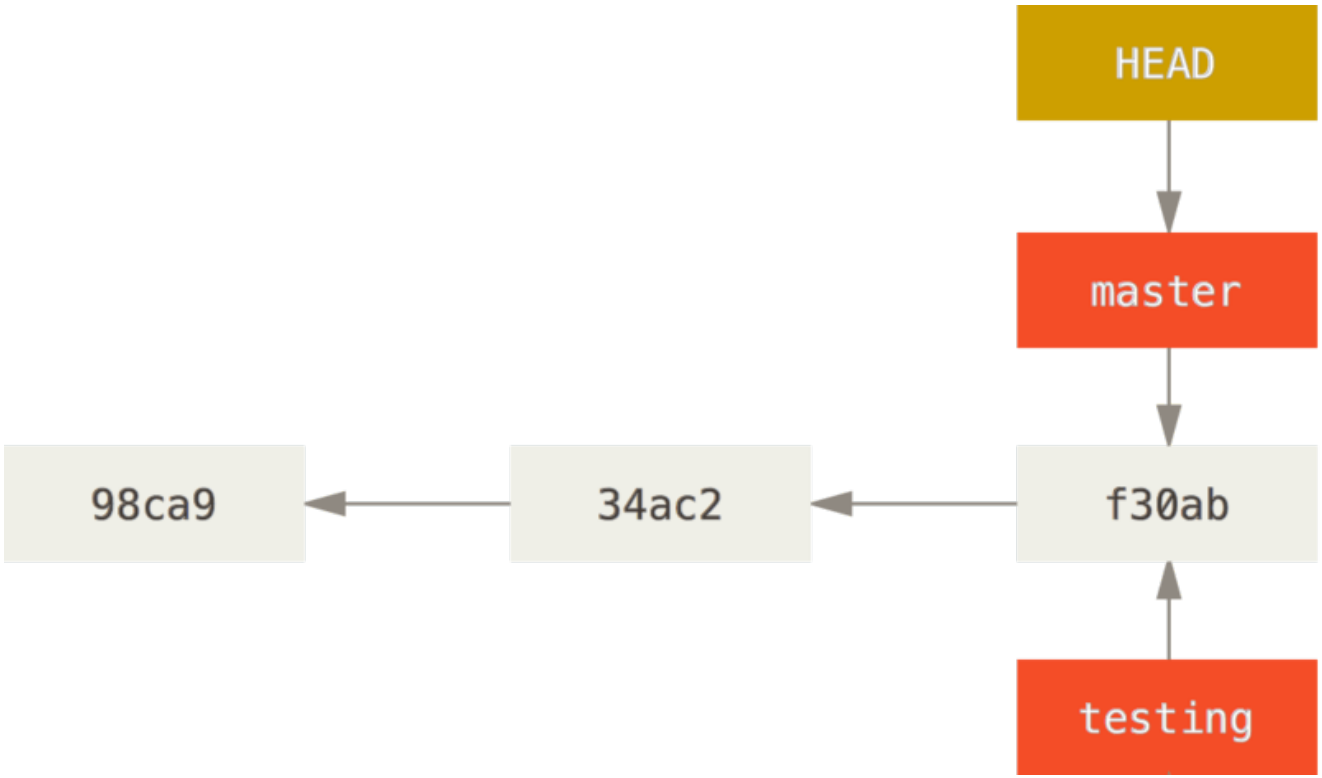
Branches let you diverge in development from the main project history without affecting it. They are good for experimental changes.

A branch in Git is a pointer to a commit.

Branches

When you make a first commit, you start on the `master` branch.

A special pointer called `HEAD` tracks which branch the working tree is on. Committing moves the current branch and `HEAD` forward one commit.



Branches - Important Commands

`git branch`

Show the current branch along with a list of all branches.

`git checkout -b <branch>` *OR* `git switch -c <branch>`

Move `HEAD` to a new branch. Uncommitted changes are preserved.

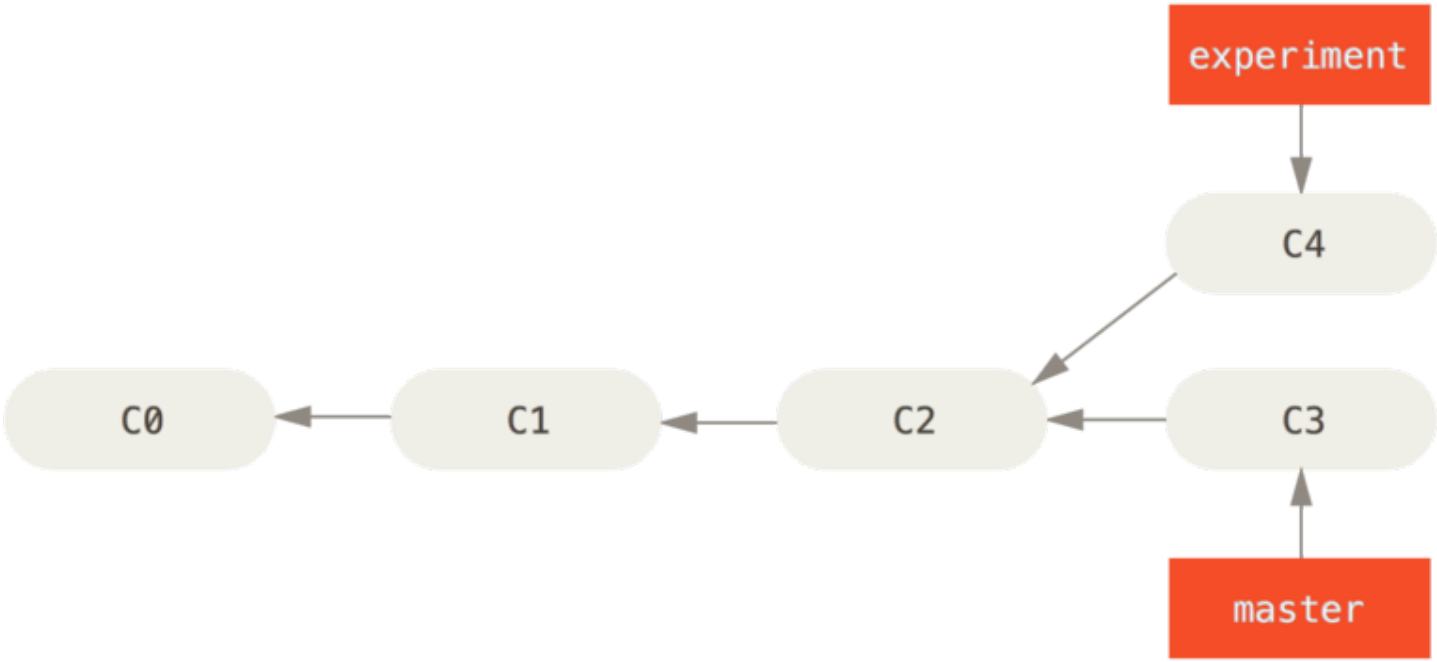
`git checkout <branch>` *OR* `git switch <branch>`

Move `HEAD` to another branch and change working tree files accordingly. Won't work if you have uncommitted changes in a file that would be affected.

Branches - Combining Branch Changes

Combine the changes of two branches with either a **merge** or **rebase**.

The working tree will look the same either way, but the commit histories will look different. Also, rebasing can be dangerous.



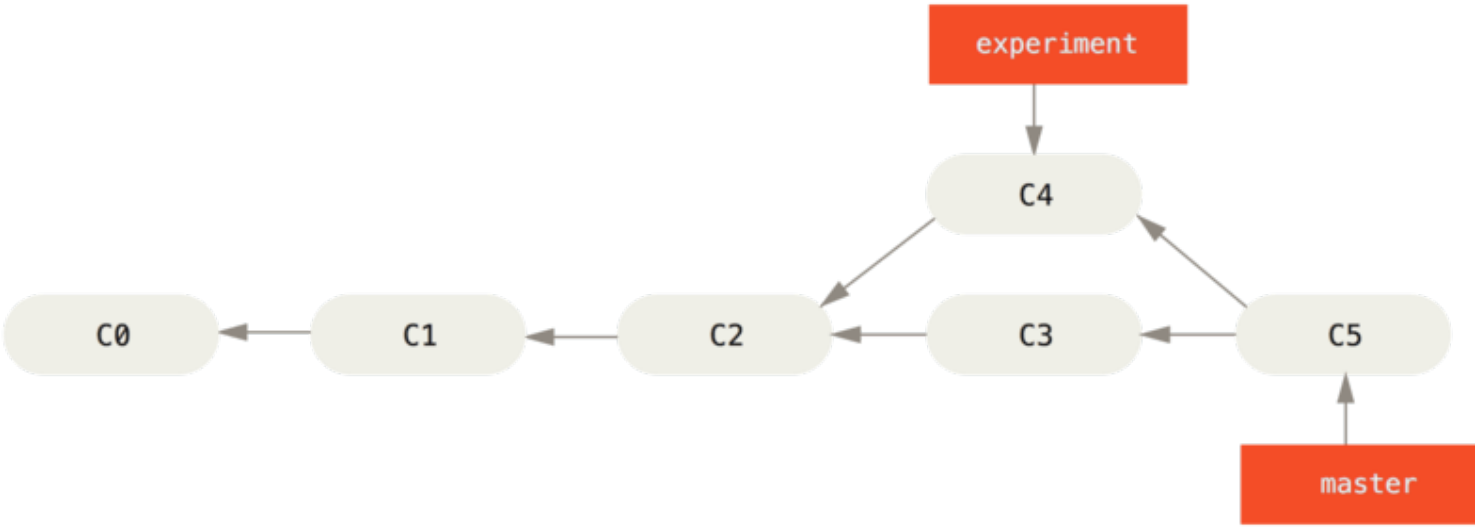
Branches - Combining Branch Changes - Merge

`git merge <feature branch>`

Create a new commit on the current branch that includes changes from the tip of the feature branch.

If branch histories don't diverge, Git will do a *fast-forward* merge, i.e. move the current branch and HEAD forward to the feature branch.

If branch histories diverge, the current branch will have a commit added containing the feature branch changes. There may be a merge conflict.



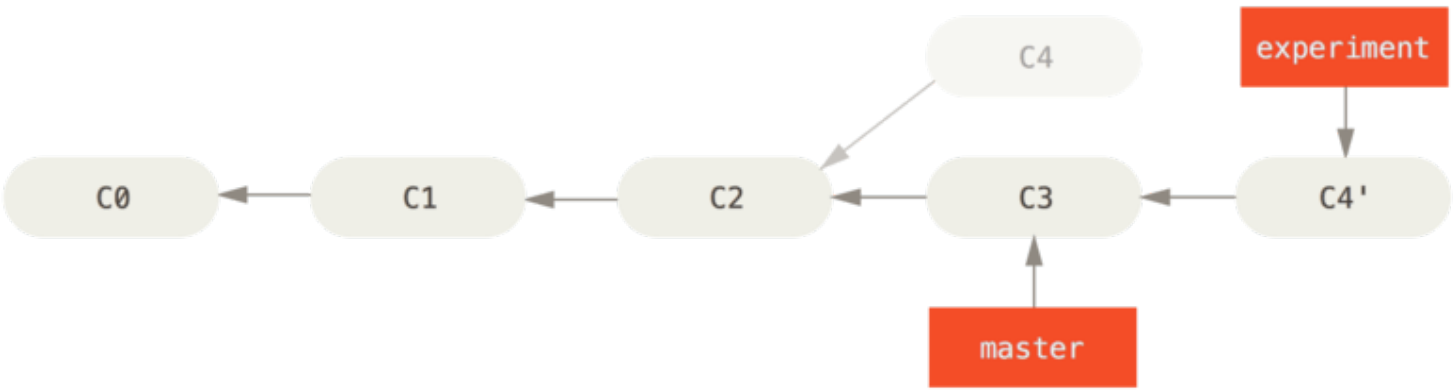
Branches - Incorporating Branch Changes - Rebase

`git rebase <base branch>`

Reapply (*rebase*) the commits of the current branch on top of the base branch.

First, HEAD moves to the tip of the base branch. Then, Git tries to apply each new commit from the feature branch in sequence. The rebased commits have new hashes, so they're different even though they contain the same changes. After rebasing you can fast-forward merge the base branch to the tip of the current branch.

If there is a conflict, Git will pause the rebase so you can fix it.

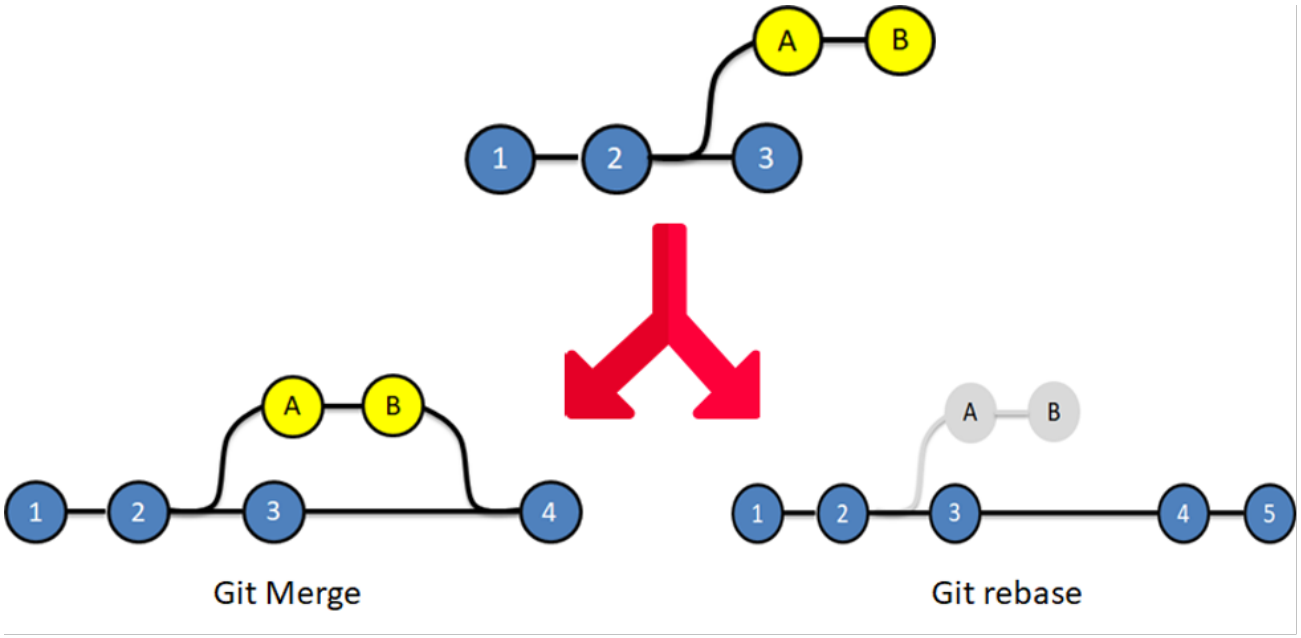


Branches - Combining Branch Changes - Merge or Rebase?

Both give a united commit history and a working tree with changes from both branches.

Rebasing gives a linear history (each commit has at most one parent), but at the cost of rewriting the history and obscuring what really happened.

Never rebase commits that are public.



Remotes

Remote repositories are hosted on the internet.

They're practical for collaboration, sharing projects, or backing up work.

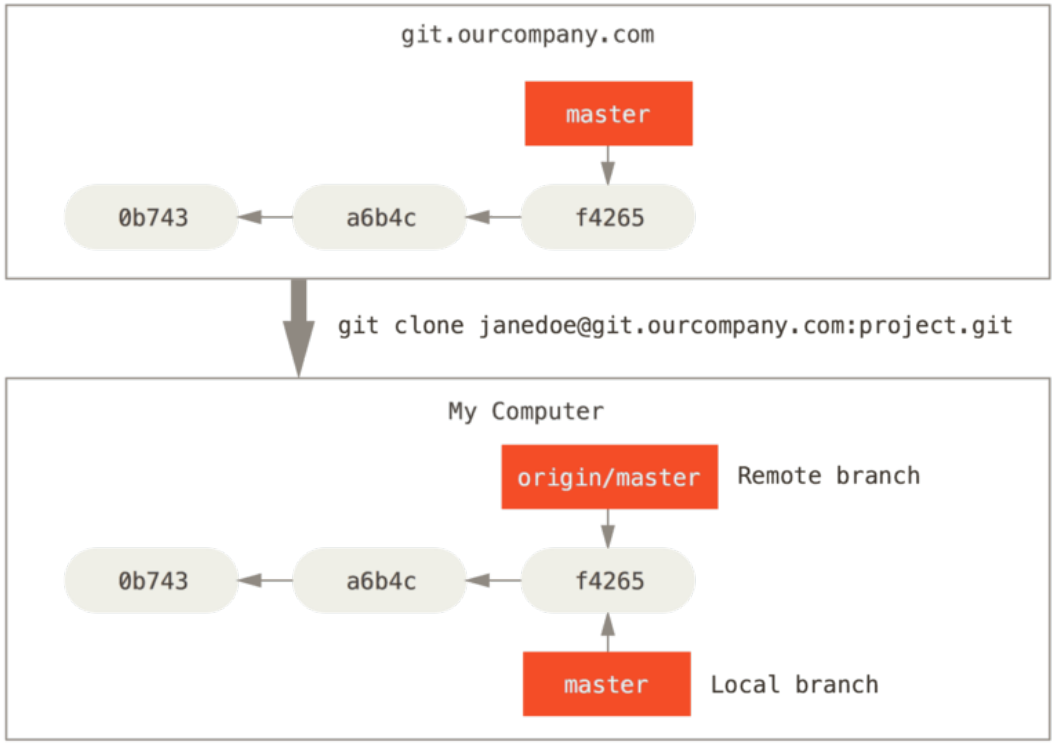
Work with remote repos through a local copy.

Remotes - Using Remotes

`git clone <URL>`

Create a copy of the repo hosted at the given URL *in a new subdirectory*.

- The URL is aliased as `origin`.
- **Remote branches** are made for every branch on the hosted repo.
- Local `master` points to wherever `HEAD` of the remote repo would be.



Remotes - Interfacing with Remotes

`git fetch <remote URL>`

Download branches and commits from the remote. The local working tree and staging index are unmodified.

`git push <remote URL> <branch name>`

Replace the given branch on the remote with the current local branch. Won't work if histories diverge.

`-f` : Push even if histories diverge.

`git pull <remote URL> <branch name>`

Equivalent to

```
git fetch <remote> <branch name>
git merge <remote branch>
```

Remotes - Important Commands

`git remote add <name> <URL>`

Add a remote alias with the given name that points to the URL.

`git checkout -b <name> <remote branch>`

Create a new local branch copied from the given remote branch.

Remotes - GitHub

GitHub provides hosting for remote repositories. Git and GitHub are not interchangeable!

GitHub also provides extra features:

- Forking
- Pull Requests/access management
- analytics
- clout
- etc...

Conventions and Etiquette

Commits and branches should be as modular as possible. Group related changes together.

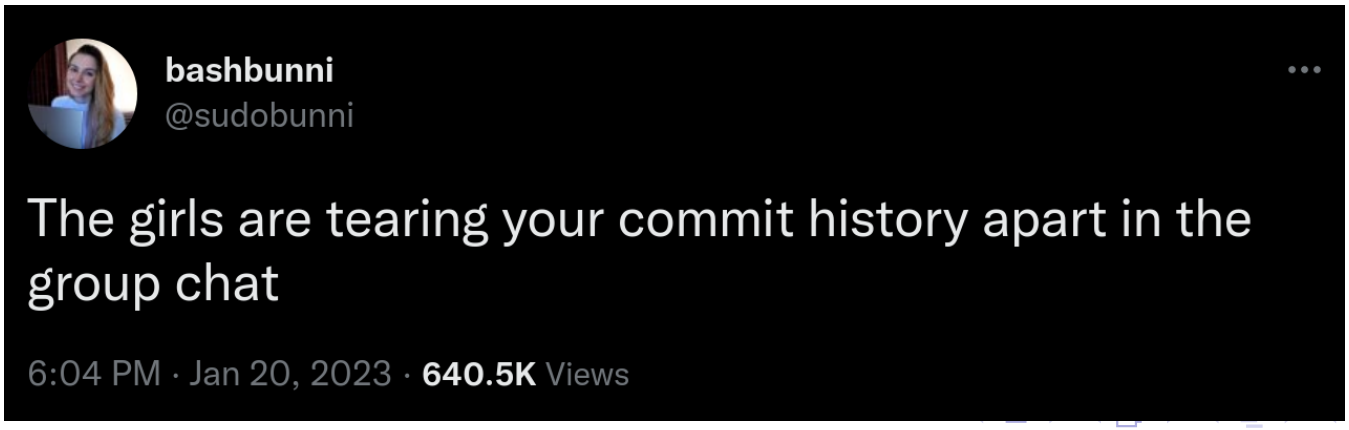
Avoid committing directly to `master` outside of merging/rebasing.

Avoid committing "compiled" files if source is being tracked already.

Commit messages should look like this:

<50 char. summary in imperative case

A blank line followed by a detailed description of what was changed, if necessary. Line wrap at 72-80 characters.



git reset

git reset <file>

The opposite of `git add <file>`.

git reset --soft <commit hash>

Move the current branch and `HEAD` to the specified commit. The working tree is preserved, and previously committed changes are staged.

git reset --hard <commit hash>

Move the current branch and `HEAD` to the specified commit. The working tree is reverted.

Other Commands

- `git reflog`
- `git cherry-pick`
- `git bisect`
- `git revert`
- `git restore`