Intro to Git

MAS.863 / 4.140 / 6.943

Agenda

- General information
- What you need for this class
- Additional techniques

General information

Best practices for this class

Important notes:

- **Do not commit giant files!** (if they can be reasonably shrunk)
    - Resize images **before** committing.
    - Compress and resize raw video **before** committing.
    - **Committed files are "forever" - and everyone has to download them!**
- Avoid the built-in GitLab "edit" button (it clutters history).

A challenge to you:

- Make your commit messages **meaningful!**
    - Imagine looking through them 6 mo from now

What?

Git: a revision control system.

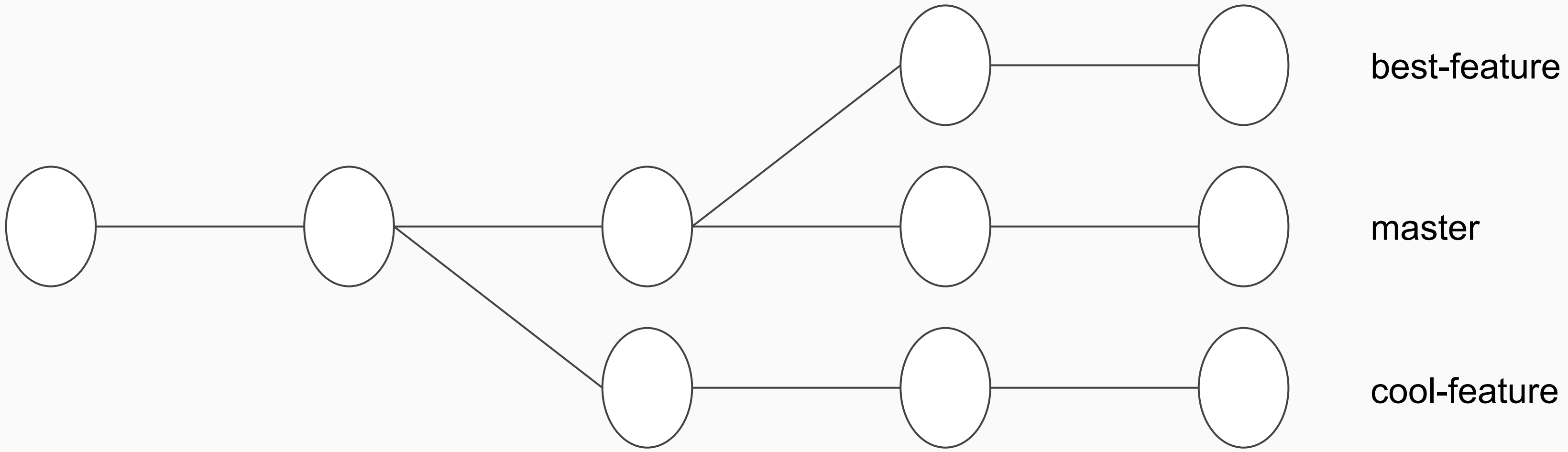
- **Complete project history!**
- Powerful **branching** and **merging** capability
- **Synchronizes** with remote repositories **on demand**

Why Git?

- Keep a detailed chronological record of **what you did** and **why**
- Easily switch between **independent feature contexts**
- **Collaborate** on source code with others in parallel
- **Resolve conflicts** that arise during simultaneous development

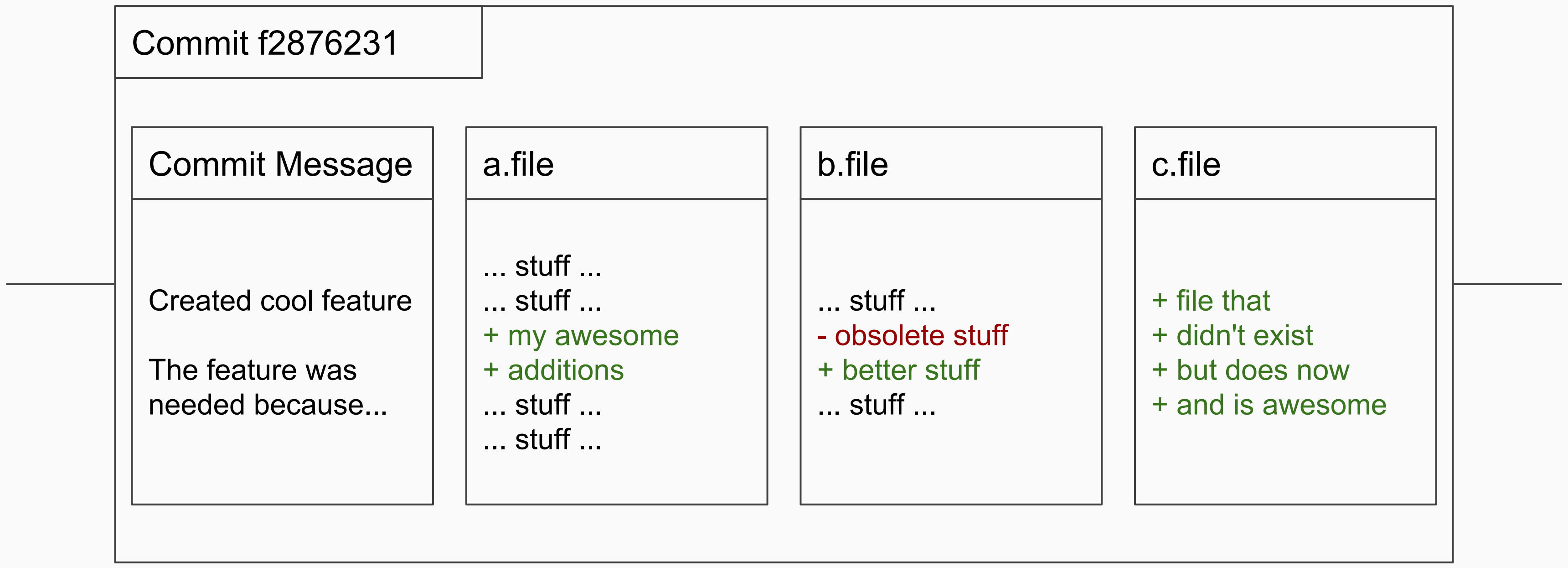
How it all works

Repositories contain **commits** organized into **branches**.



How it all works

A **commit** contains a **set of changes** as well as a **commit message** explaining what was done and why.



What you need for this class

Creating a new repository

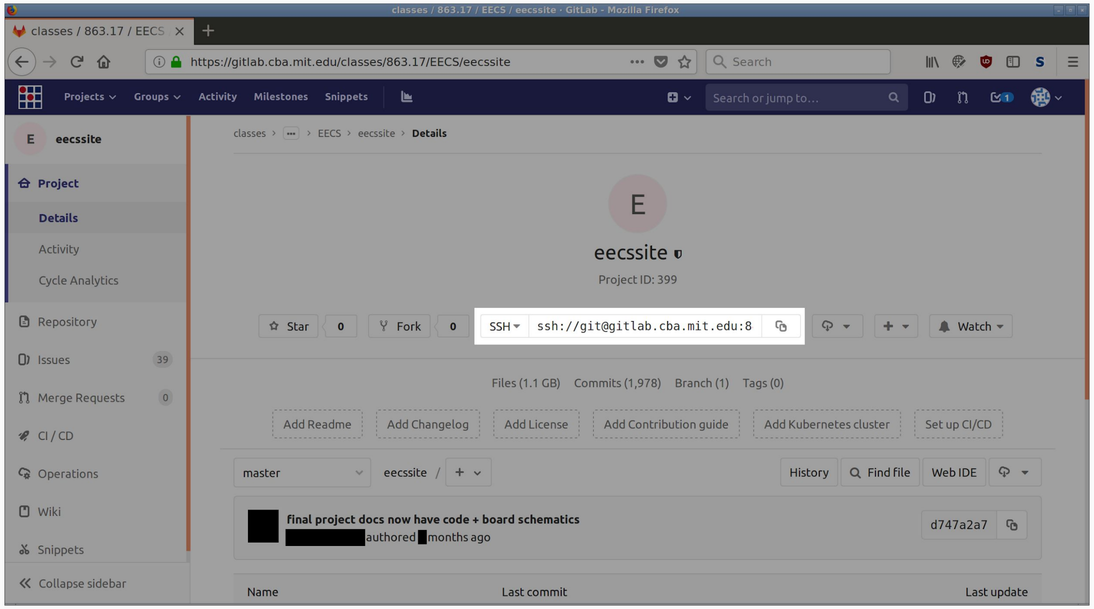
```
git init
```

Creates a blank repository in your working directory.

Cloning an existing repository

`git clone url-of-repository`

(Set up SSH keys in GitLab first! Then use the SSH clone URL.)



https://gitlab.cba.mit.edu/classes/863.17/EECS/eecssite

```
ssh://git@gitlab.cba.mit.edu:8
```

final project docs now have code + board schematics
[redacted] authored [redacted] months ago

Creating and adding SSH keys

SSH keys **identify** your computer.

- `ssh-keygen -t rsa -b 4096`
  creates a new key.
- `cat ~/.ssh/id_rsa.pub`
  prints your *public* key to the terminal output.
- Copy your public key into GitLab (Settings -> SSH Keys)

Creating a branch

By default a Git repository contains one branch called "master".

- `git branch my-awesome-branch`
  creates a new branch called "my-awesome-branch"
- `git checkout my-awesome-branch`
  switches to that branch

Committing your work

First do some work. Then:

- `git add file1 [file2] [...]`
  stages changes in file1, file2, etc. for commit
- `git reset HEAD file1`
  unstages all changes in file1
- `git status`
  reviews what files you have and haven't staged
- `git diff --staged`
  reviews exactly what changes you've staged
- `git diff`
  and what changes (to existing files only) you haven't

Committing your work

- `git commit`
  commits those staged changes to the current branch
  after asking for a **commit message**

A good commit message contains:

- a short (one line) summary of what you did
- a long (~paragraphs) description of **what** you did, **how**, and **why**
  - what problem did it solve?
  - what techniques were used?
  - what pitfalls are to be avoided?

Reviewing history

You want to understand what's been done in the past.

- `git log`
  shows commit messages for the current branch
- or use GitLab

Updating a commit

You realize you want to update a commit (either message or content), and you *haven't uploaded it yet.*

- Stage any file changes you want to include.
- `git commit --amend`
  will ask for edits to the commit message
  and bring in any staged changes.

  This creates a new commit with the same parent,
  and makes the current branch point there.

Uploading your work

Repositories can be linked to "remotes". Cloned repos have a remote named "origin".

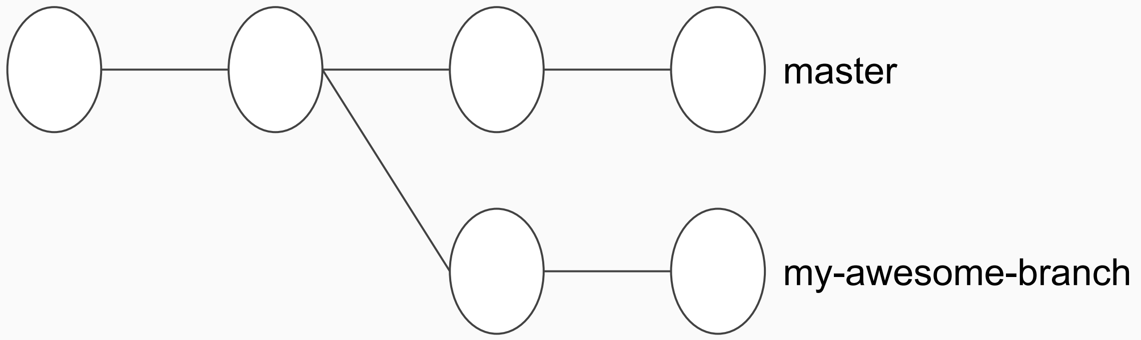
- **`git push origin name-of-branch`**
  attempts to update the default remote with your work on the named branch

Keeping track of remotes

You want to bring your work into **`master`** branch because that's what's deployed to the website. First:

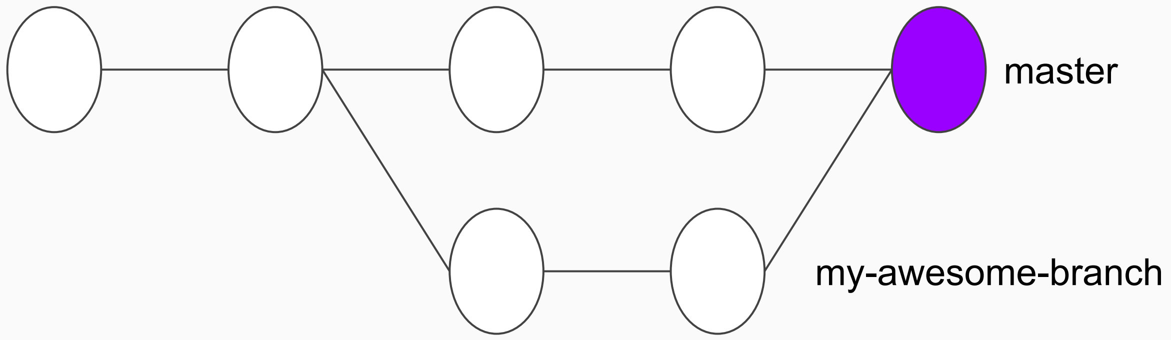
- **`git checkout master`**
- **`git pull`**

  merges in the remote changes to your local master branch



Integrating your work

- `git merge my-awesome-branch`
  merges in the work from my-awesome-branch into current branch
- fix any **conflicts** when Git complains
  - edit files by hand
  - `git commit`
- `git push origin master`
- if this doesn't work, `git pull` and try again



Best practices for this class

Important notes:

- **Do not commit giant files!** (if they can be reasonably shrunk)
  - Resize images **before** committing.
  - Compress and resize raw video **before** committing.
  - **Committed files are "forever" - and everyone has to download them!**
- Avoid the built-in GitLab "edit" button (it clutters history).

A challenge to you:

- Make your commit messages **meaningful!**
  - Imagine looking through them 6 mo from now

Other GitLab tools

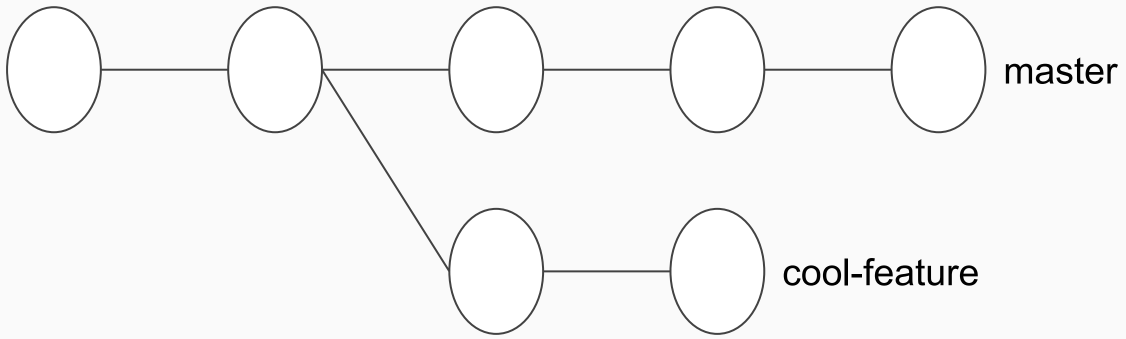
- **Issue tracker**: track tasks, communicate within sections.
- **Kanban board**: visualize issues within a workflow.
- **Labels**: categorize issues by type, severity, importance, etc.
- **Milestones**: group issues into progress checkpoints.

Additional techniques

Rebasing

You're working on a branch that you *haven't pushed yet* and master has updated in the meantime.

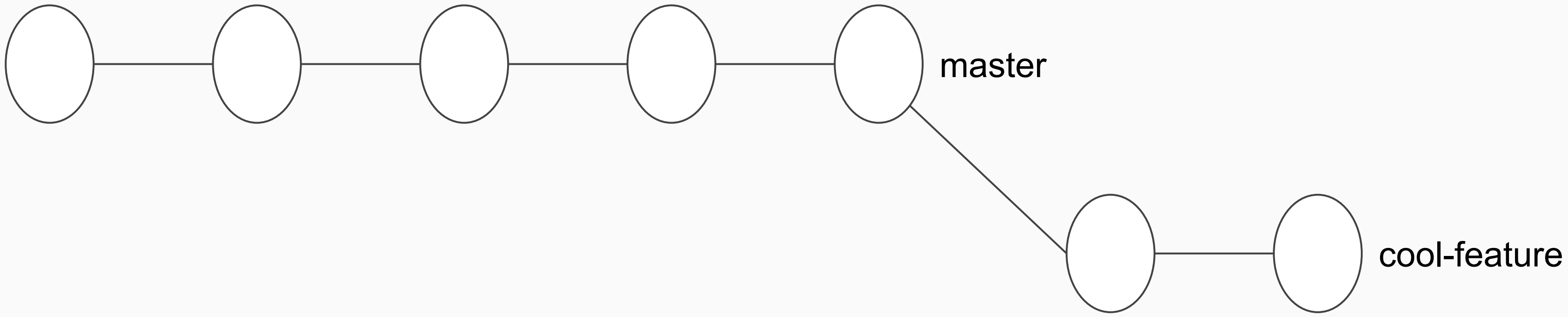
You want to bring in the new changes from master and keep working on your feature branch.



Rebasing

```
git checkout cool-feature
git rebase master
```

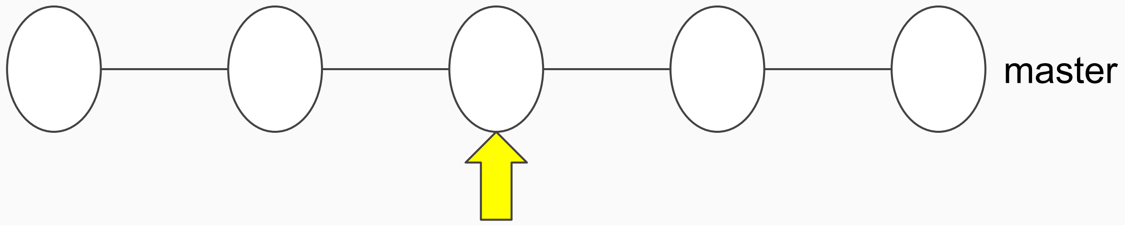
moves cool-feature to start from the most recent commit in master.



Checking out a specific commit

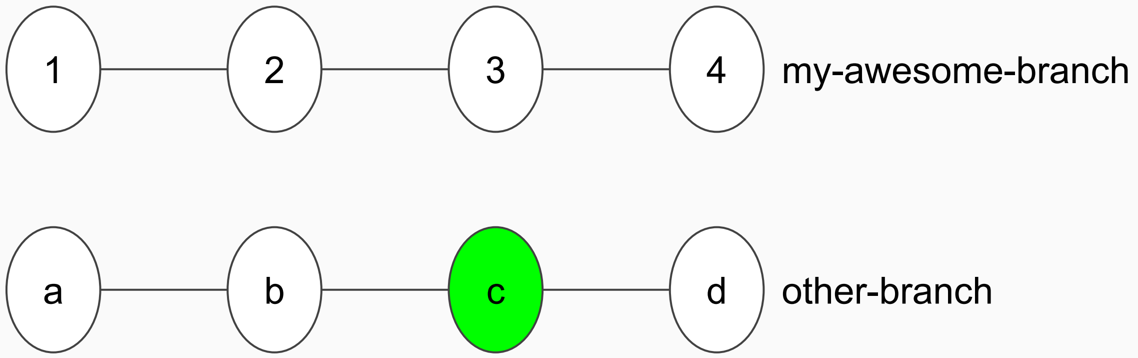
Sometimes you need to see the repository at a particular point in time.

- `git checkout <commit-hash>`
  will check out that specific state.
- You can do whatever you like!
  - look around, make changes, even make commits...
- `git checkout -b <new-branch-name>`
  will save any new commits you made on top to a new branch.



Cherry picking from a different branch

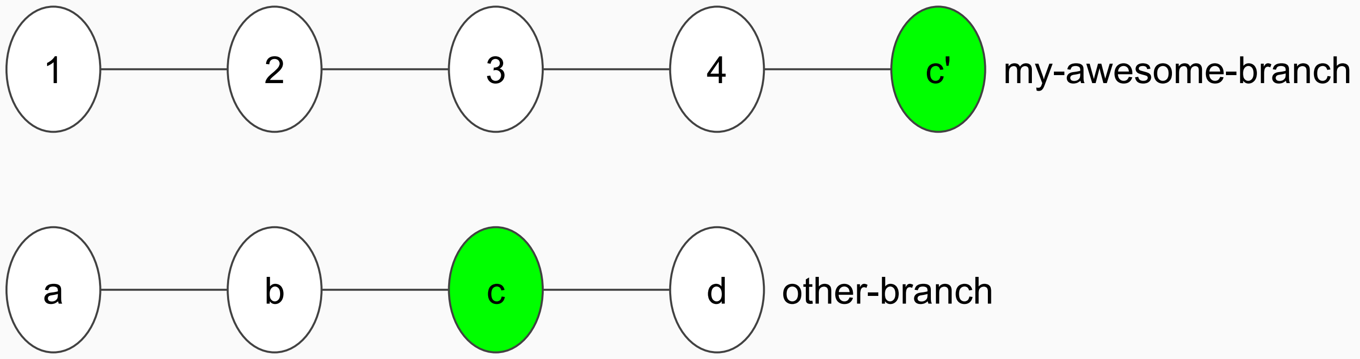
While working on a feature branch, or looking at someone else's work, you realize you really want to bring in a specific commit from a different branch.



Cherry picking from a different branch

`git cherry-pick <hash-of-commit-c>`

replays commit c on top of your current branch.



Recovering "lost" commits

There's no such thing as a "lost" commit!
If you commit your work, it lives in the repo "forever".

- `git reflog`
  lists every commit you made recently (even if e.g. its branch is gone)