Week 5

Announcements

- Basic, Advanced 4 out
- Projects are out now!

Lecture 5: git gud

"Boy I sure do love creating a merge commit every time I pull!"

Overview

- Review
- Rewriting history
  - Fixing mistakes
  - Cleaning up
- Stashing
- Understanding remotes
- Workflows
  - Overview
  - Centralized workflow
  - Merge conflicts

Review

- `git init`
- `git status`
- `git add`
- `git reset`
- `git checkout` (`git restore`)
- `git commit`
- `git status`
- `git branch`
- `git checkout` (`git switch`)
- `git merge`

Review

- `git remote`
- `git push`
- `git pull`

Review

- Creating a local repository
- Staging files
- Making commits
- Making branches
- Merging branches
- Interacting with remotes

Rewriting history

- This section involves changing up commit history
- Use with caution if you have already pushed them to a branch that's shared with other people
- If you force the remote branch to take on the rewritten history, it'll cause the other peoples' local branches to be incoherent

Fixing mistakes

Scenario: made a commit by accident and want to "uncommit"

- `git reset` to the rescue!
- `git reset HEAD~1` is a common internet answer given without explanation

Dissecting it:

- `git reset` sets the HEAD to a specified state (bringing the current branch along for the ride)
- `HEAD~1` specifies that we want the HEAD to take on the state of the commit that is 1 before the HEAD
    - We could provide the commit's hash or some reference/pointer to a commit instead (e.g. branch or tag name)

Fixing mistakes

Scenario: made a commit by accident and want to "uncommit"

- `git reset` has three major modes in this application:
  - `--soft`: undoes the commit, leaves the Working Directory untouched, and leaves the changed files **staged**
  - `--mixed`: default, undoes the commit, leaves the Working Directory untouched, and leaves the changed files **unstaged**
  - `--hard`: undoes the commit and brings the Working Directory to the state of the commit, discarding the changes

Fixing mistakes

Scenario: forgot to add a file

- `git add <file>`
- `git commit --amend`
- `git commit --amend` will bring the currently staged changes into the current commit and allow you to edit the commit message
  - If you forgot to *delete* a file, just `git rm <file>` to remove and stage the removal, the `git commit --amend`
  - `git commit --amend --no-edit` won't ask you to edit the commit message

Scenario: typo in commit message

- `git commit --amend` with no staged files will just have you edit the commit message

Cleaning up

- `git rebase` is an incredible powerful command that allows you to rewrite history
- `git rebase -i <base tip>` is the form you most likely will use
  - `<base tip>` is either a commit hash or branch that you want to replay commits onto

Common use-cases include:

- "Squashing" commits
  - This allows you to put multiple minor commits into a single more substantial one
- Reordering commits
- Rewording commits
- Playing back commits on top of another branch (more on this later)

Caveats of rewriting commits

- `rebase` and `commit --amend` *rewrite* commits if you make changes
- A commit's hash depends on the files' data, the commit message, commit info, **and the parent**

What does this mean?

- A reworded commit is technically a *new* commit
- Adding a file to a commit turns it into a *new* commit
- **Child commits will techincally become new commits**
- Don't believe me? Check the hashes
- **Be wary if you have rewritten commits that have already made it to a remote branch that other people access**
- **If you force the remote branch to take on your rewritten history, people who have previously pulled to their local repositories will no longer have coherent histories**

Stashing

- `git stash` allows you to save the state of your Index **and Working Directory** into the "stash" (acts like a stack) and rolls you back to a clean Working Directory
- This is particularly useful if you need to jump around different branches while you have some modified files hanging about that would be changed by the other branches
  - `git checkout <branch>` won't let you switch branches if the target branch modifies files that are currently already modified
- `git stash` will implicitly perform a `git stash push` and putting your current Index and Working Directory's state onto the stash's stack
- `git stash pop` will bring the top entry of the stash's stack into your Index and Working Directory, deleting that entry from the stash
- Note how `pop` will delete automatically delete that entry
- `git stash apply` will do the same thing as `pop` but without the automatic entry deletion: useful in case the `pop` fails/has issues
- `git stash drop` will then delete the item at the top of the stash's stack

Understanding remotes

- Recall that a **remote** is repository hosted on some server
- Recall that remote names have no special meaning: `origin` is just the default when you clone
- A remote has its own set of branches and commits, being another copy of the repository in this distributed system
- When working locally, note that `master` is not the same as `origin/master`
  - `master` is an arbitrary local branch with that name that may or may not be "tracking" `origin/master` (we'll elaborate later)
  - `origin/master` is a branch named `master` on the remote `origin`, which could for example have a URL of `git@gitlab.umich.edu:eecs201/somerepo.git`

Understanding remotes

- This is where we get the legendary command `git reset --hard origin/master` to undo everything locally
  - Maybe we have totally borked our local branch `master`: maybe its history has been destroyed by a rebase and we just want to go back to something sane
  - This does a hard reset for `master` using `origin/master`'s commit as the target state

Tracking branches

- They may have the similar names, but `master` is a local branch that is *tracking* `origin/master`
- What this means is that `master` looks at `origin/master` as the place to push/pull commits to/from
  - `origin/master` is known as its **upstream branch**
- If you checkout a branch that exists on *one* remote but not locally, Git will automatically create a local branch of the same name and have it track that remote branch.
  - This only works if there is only a single remote with that branch name.
- We can arbitrarily create tracking branches of arbitrary names that track remote branches
  - `git checkout -b top-of-tree origin/dev` will create a local branch `top-of-tree` that tracks `origin/dev` (and switch the current branch to `top-of-tree`)
  - `git branch -u origin/issue149` will cause the current local branch to track `origin/issue149` (i.e. setting the upstream)

Tracking branches

- Your local repo does keep a cached copy of `origin/master`, which gets updated whenever you `git fetch`: it doesn't automatically keep in touch with the server
- `git pull` performs `git fetch` then merges `origin/master` into `master`
  - Many a Git beginner has been victim to this automatic merging (me included)...
  - There's also rebase mode where it rebases `master` onto `origin/master` instead

Workflows

Maybe you have run into this scenario...

- You are working with a group of people on a project and decide to use Git to collaborate, and host your repo on the campus GitLab
- Perhaps none of you are particularly versed in Git (with one member even opting to use the GitLab webpage to upload/edit files!)
- So each of you do your work, `push`ing and `pull`ing to `master`

Workflows

Maybe you have run into this scenario...

- Almost immediately you're going to run into a situation where two (or more) people race to push their commits
  - Person A pushes their commit first
  - Person B tries to `push`, but the server refuses and tells them that their local branch is behind
  - Person B `pull`s, causing a superfluous merge commit between `origin/master` and Person B's `master` branch
  - Person B then `push`es their original commit and a merge commit that has the amazing default message that says `master` got merged with `origin/master`...
- As the project continues, each person is pushing tiny incremental commits.

Workflows

- Workflows give a structure to how we should perform our versioning work
- Git does not explicitly lay out workflows for us to follow
- This lecture we'll be focusing on what Atlassian would call a "Centralized Workflow"
  - I've chosen this as it's fairly standard and is manageable and suitable for school life, while giving you the fundamentals
  - Read more about Workflows in the Atlassian tutorials

(Basic) Centralized Workflow

- In this we have a main branch that code is being contributed to (e.g. `master`, `dev`)
- For brevity, let's refer to the remote as `origin` and the main branch as `dev`
- Locally each user tracks `origin/dev` on some local tracking branch (e.g. `dev`)
- Each user works on this local tracking branch on their feature/fix
- The user makes the commits they want
- The user uses `git rebase` to squash, reorder, and reword commits to package up their feature/fix more nicely
  - Probably a good idea to squash two commits where one has a sizeable change and the other fixes a typo in the first
- The user then pushes their change

(Basic) Centralized Workflow

- If it fails due to the local branch being behind, then the new commits need to be pulled
  - `git pull --rebase origin dev` will perform a rebase of your new commit on top of the commits fetched from `origin` instead of a merge, avoiding the merge commit
  - Local branch `dev` will be fast forwarded to `origin/dev`, and your new commits will be put on top of `dev`'s new up to date spot
- Now the user should be able to `push` (if they can't due to some other speedy user, they simply just have to do another `pull`)
- As a result, we now have a relatively clean history with meaningful commits free of *"superfluous"* merge commits

+Feature Branching

- The idea behind this is to have a main branch (e.g. `dev`) represent a stable, passing codebase
- Feature branches are spawned off, have their features completed and committed, and have their commits *brought back* into the main branch
- Feature/topic branches could range from actual remote branches with multiple contributors to a single person handling their issue locally
- The flow is very similar, with feature/topic branches that have multiple contributors having something like a miniature Centralized Workflow
    - When the feature is complete (and tested), it can be locally `merge`-ed into `dev` and pushed to `origin/dev`
    - This merge commit will capture this branching and merging behavior in the history
    - Alternatively if you want to have a linear history, you can use `rebase` instead of `merge`
    - One option at this point is to have a person responsible for bringing feature commits into the main branch

+Feature Branching

- You can also do this locally
- Say you were assigned bugs 1, 2, and 3
- You have a local tracking branch `dev` that tracks `origin/dev`
- You then have three local feature/topic branches `bug1`, `bug2`, and `bug3`
- You can perform your fixes for each of them, switching between them when you get stuck, etc.
- When you finish up bug 2, you can get the latest changes for `dev` and then rebase/merge `bug2` onto/into the newly updated `dev` and perform the appropriate push
- You then repeat this process for bugs 1 and 3

Merge conflicts

- Sometimes when you perform a `merge` or `rebase` the commits of one branch conflict with the commits of another
- This is called a "merge conflict"
- The `merge` or `rebase` process stops, allowing for you to edit the files that have conflicts to get the file to have the correct contents
  - This conflict resolution stage will insert some special strings into your code saying that one branch/commit had these particular contents and another branch/commit had these certain other contents.
  - You might've seen `<<<<<<<`, `=======`, and `>>>>>>>` at some point
- When you finish up with the conflict resolution, stage the necessary files and finish the `merge`/`rebase` procedure
  - `git status` will tell you the appropriate command to run to continue

Demo: Centralized and Merge Conflict

1. Alice makes initial commit, creates a mainline `dev` branch
2. Bob clones and checks out the `dev` branch
3. Alice adds a contributors block
4. Bob pulls the new changes
5. Bob updates the contributor block and pushes
6. Alice pulls the new change
7. Alice and Bob both change lines in the same area
8. Alice pushes first
9. Bob tries to push, gets rejected
10. Bob does a rebase pull
11. Bob deals resolves a conflict

Workflows

- This was only a ***taste*** of workflows
- There are different kinds
- You may develop your own style of local workflow as you get more used to Git
  - The Centralized Workflow and its kind are more of remote collaboration workflows
  - You don't have to go *strictly* by the local workflows described here
  - Git is very flexible by nature, so workflows themselves aren't really built into the tool

Now go forth!

You are officially dangerous with Git :)

(There's more stuff, like `git cherry-pick`, `git blame`, and `git bisect`!)

Questions?