Git and Gitlab

Note: start by creating a new folder to work in so you don't affect actual files.

- Getting Started

  - Creating a repository and adding files

Info:

Note: when you first try this, git will ask you to set your identity. Follow its instructions and set your global identity. You can also set a local id to separate work and personal projects or similar. Local overrides global config

Try:

Initialise a repository in your test directory (git init)

Create some files using your favourite editor

Add them (git add {filenames}) and commit them (git commit)

Try (git ls-files) to list all the files git is controlling

- Writing a Good Message

  - How to write a commit message

Info:

You already committed some files, but probably used a message like "test" or "first commit". Git is very good at telling you what files and code changed, but that doesn't explain the function that changed. For example "changed email" is a lot less useful than "Updated contact email to Jim (new administrator)"

Try:

Edit one of your files to add something, and commit it with a "good" message. Remember the first line is the title, then a blank, then a message body

Pick an interesting repository from e.g. https://github.com/trending and read some of their messages (go to the repository and click on where it says {x} commits) Are they any good? Do they explain why as well as what?

- Viewing State and History

  - Using the log and the status

  Info:

  If you have a lot of object files or local files you don't want in the repo, you can have git ignore these completely by creating a .gitignore file (https://git-scm.com/docs/gitignore) This means they aren't show in the status output

  Try:

  Try git status. Now change and add a file (don't commit) and try status again

  Create some files (don't add them) and try status

  Create a very simple .gitignore. Object files should be ignored; on Mac .DS_Store should be included as should any temporaries created by your editor or IDE

  View the log (git log). Note your commit messages and the name/email you setup in the first section.

- Committing Everything

  - What git commit -a includes and excludes

  Info:

  Commit -a commits all changed files that git knows about, whether added or not. But it doesn't commit any files that were *never* added

  Try:

  Try git commit -a. After the last section you should have some added files, some un-added files and some ignored files. Which are committed?

  If you made the .gitignore, try adding something which is ignored. Note git will complain and tell you to use force

- Viewing and Undoing Changes

  - Changes can be viewed and reverted

  Info:

Remember, git history is intended to be preserved. You *can* change it, but if you shared with anybody else, you can make a real tangle, so be cautious

Try:

Make a change to a file, and commit it. Revert the commit (git revert)

The file should have gone back to the unchanged version

View the log (git log) and note the original commit and the revert

(Bonus) : make another change, commit that, and revert the previous 2 commits. The log will now show you reverting a revert. This can go on forever

- Basic Branching

- Creating a branch

Info:

Branches can be created from branches, but mostly you will want them to be based off master. That means you should be on master when you create them

Try:

Create a branch (git branch {branchname})

Move to the new branch (git checkout {branchname}) and commit something

Return to master, and check your change is not there

Show all branches (git branch) and note the * telling you your current branch

- Basic Merging

- Merging a local branch

Info:

Simple merges pull another branch into your current one. You can do this either way (a into b, or b into a) but make sure to get the right way round!

Try:

Merge the branch you just created into master (git checkout master, git merge {branchname}) This should fast-forward smoothly

Create a new branch. Change both master and the branch on the same line, but differently. This should create a conflict

Try to merge master into your new branch. You should get a conflict

Resolve conflicts. Open the troublesome file and search for <<<, ===, >>> Combine your changes from the two blocks to create one working version and remove the markers

Finish the merge, either by committing or by continuing (git merge continue). If you missed any markers, git will refuse

- A Basic Patch

- Patches let you share changes without a remote

Info:

Patches are used extensively for the Linux Kernel. They are useful for small or temporary fixes. You can skip this or leave it for later

Try:

Make a new branch off master to freeze the current state

On this branch, make a new commit

Make a patch from this commit (https://git-scm.com/docs/git-format-patch)

On master, apply the patch and commit (https://git-scm.com/docs/git-apply)

Use the IDs of the two commits to confirm (git diff) there are no differences

- Using a Remote

- Getting code from a remote

Info:

This uses Gitlab for simplicity, but you may use other servers. This part of things doesn't change if you do

Try:

In a new, empty, non-git directory, clone EPOCH. Either pick something interesting or use our Examples repo. Find the green "clone or download" button and copy the https address. Run git clone --recursive {address}

Run git log to see recent activity

Check whether there are any branches with git branch or git branch -a

Commit something, then try git status

- Git Tags

  - Cloning a tag

Info:

Tags are very useful, and quite simple to use. For EPOCH they're used to create the version numbers

Try:

List tagged versions with git tag

Pick a version number and examine the commit its attached to with git show {tagname}

Checkout the version into a branch using git checkout {tagname} -b {new branch name}

If the version you picked is old, you may see that the SDF submodule gets out of sync:

```
Petunia:epoch heatherratcliffe$ git checkout v4.9.4 -b vers_4.9
Checking out files: 100% (266/266), done.
M   SDF
Switched to a new branch 'vers_4.9'
```

This is saying that the SDF code has been modified since this tag. git submodule update --recursive will wind this back to how it was then. When you checkout a newer branch again, you run it again to re-sync things.