Chapter 1: Introduction

Git is basically just a **version control system**, which means it **tracks and manages changes to files over time**.

1.1 Git Installation (Windows)

Windows only comes with Command Prompt that is not Unix-based, whereas Linux and Mac have a default shell CLI called **Bash**. To experience Bash on a Windows machine, need to install **Git Bash** that comes with Git.

i) Go to https://git-scm.com and download the 64-bit version.
ii) Change Vim editor to VSC.
iii) Type **git --version** to test if everything went right.

1.2 Unix Commands (use Git Bash for Windows)

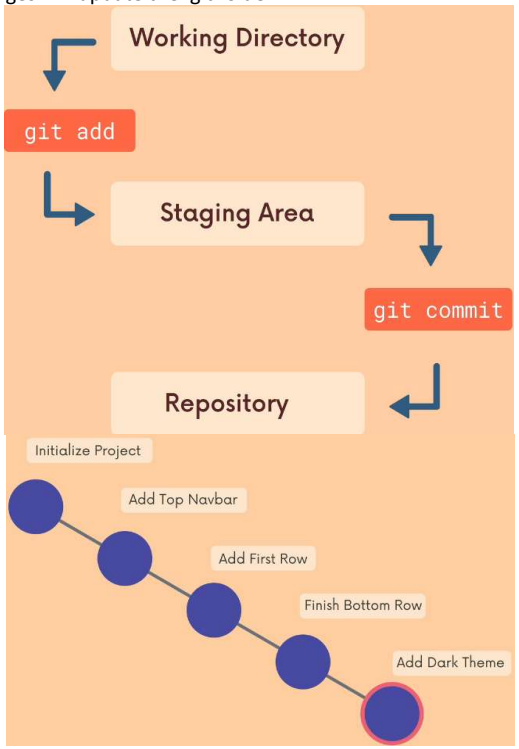
	Windows	Mac
List all files and folders	ls ls -a (hidden files)	Ls ls -a (hidden files)
Open File Explorer GUI	start .	open .
Explore a Folder	ls FolderName start FolderName	ls FolderName open FolderName
Present WD	pwd	pwd
Change Directory	cd FolderName cd ..	cd FolderName cd ..
Create a new file (can multiple)	touch file1.txt file2.py file3.pdf	touch file1.txt file2.py file3.pdf
Create folder	mkdir Folder1 Folder2	mkdir Folder1 Folder2
Remove files (rf means recursive + force)	rm file1.txt rm -rf Folder1	rm file1.txt rm -rf Folder1

1.3 General Configurations

- See your current username: **git config user.name**
- Configure your username associated with your work (can change many times. This is just a name you can attach to your commits and has NO IMPACT and NOT RELATED on your actual GitHub account): **git config --global user.name "Tom Hulse"**
- Configure your email (same as above): **git config --global user.email** myemail@gmail.com
- Change from Vim to VSC editor (see git website for others): **git config --global core.editor "code --wait"**
  - If you met an error where VSC doesn't know what "code" means, then go to VSC, press Ctrl+Shift+P (command palette), type in ">code", then try committing again.

1.4 Git Terms (Basics)

- A **Git repository** is a workspace which tracks and manages files within a folder. Each project = 1 git repo.
- When you type *git init*, it creates a hidden .git folder with many subfolders within it. **This is the repo itself**. Git will track any changes inside the current directory it is at AND all its children. Any changes will update the .git folder.



- Each of the above checkpoints is actually a **commit**. But before committing, there is an intermediate step of selecting out what things you actually want to commit by **adding** to a **staging area**.
  - This allows you to group your checkpoints into meaningful and sensible commits, eg first commit is for work on People, while second commit is for work on Chatbot etc.
  - This is why you must *git add* before doing *git commit*.
- Steps on making a commit:
  - Make some changes to some files.
  - Type *git status*. Notice how it points out **untracked files**.
  - Add them to staging area with *git add file1 file2.* If you type *git status* again, it will show (i) what are changes to be committed, (ii) what remains untracked, and (iii) any modified files that are not committed.
  - Commit with *git commit -m "simple changes"*. Then type *git status* again and you will see "working tree clean" which means everything in your working directory is tracked and is fully 100% synced with the .git folder which is now up to date.
  - Pro tip: use present-tense and in imperative order (as if you are giving orders) when writing your commit messages.

```
$ git log
commit d140c5cf7c693dffa65daecd4369690a2047f455 (HEAD -> master
Author: xuanyucodes <a170071@e.ntu.edu.sg>
Date:   Tue Jun 7 01:27:18 2022 +0800

    begin work on chapter 1

commit 623d9f02011a05d83e422d3c35cbec4a5c610616
Author: xuanyucodes <a170071@e.ntu.edu.sg>
Date:   Tue Jun 7 01:21:36 2022 +0800
```

- When you type *git log*, you will see the **commit hash** etc.
- Pro tip: do **atomic commits**! This means that each keep should be focused only on a single thing. This makes it easier to undo or rollback changes later on, and easier to review.

Git Ignore

- Create a **.gitignore** file so you will never commit such files (secrets, API keys, credentials, OS files, log files, dependencies and packages).
- Typically place at the root of the directory but not necessary.
- Ways to ignore:
  - .DS_Store will ignore files named .DS_Store
  - folderName/ will ignore an entire directory
  - *.log will ignore all files with the .log extension
  - See more methods in docs.

1.5 Git Basics

General Maintenance

- Current status of a git repo: **git status**
- See history: **git log**
  - Many other options. See docs.
  - Shorten logs: **git log --oneline**

Initialisation

- Create new git repo (which is the hidden .git folder!) in the current directory you are at: **git init**

Committing

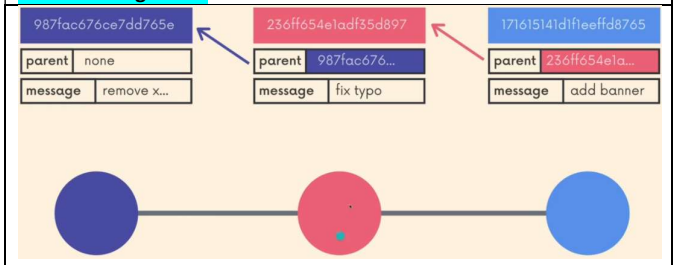
- To add files to staging area first: **git add file1 file2 file3**
- To commit things in the staging area (best to add message here if not an editor will pop up): **git commit -m "my message"**
- To stage all changes at once: **git add .**
- Stage AND commit: **git commit -a -m "blah blah"**

Amending Commits (WORKS ON THE PREVIOUS COMMIT **ONLY!!!**)

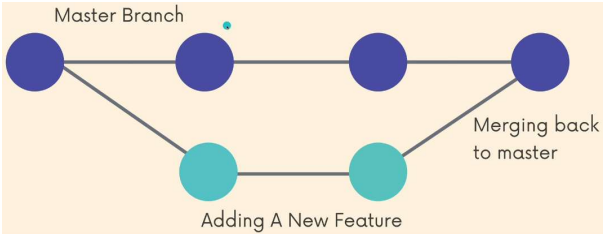
- After your first "git commit -m 'some commit'", you need to git add the new files.
- Then you type: **git commit --amend**

Chapter 2: Branching

2.1 Branching Basics

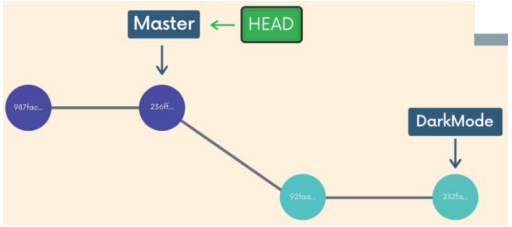


- Each commit has its own **commit hash** and references at least one parent commit before it.
- Branches allow us to collaborate in large projects easily, where we can create separate contexts to try new things, or even work on multiple ideas in parallel. If we make changes on one branch, they do not impact other branches (until you merge the branches).
- The **master** branch (sometimes called **trunk**) is just the branch you start off with – it doesn't mean anything special, it is just a normal branch. You can delete it, rename it etc. But most people treat it as the "official branch"; this is up to you to decide.
- The **master vs main** issue is because "master-slave" is offensive to some people. Git still calls this branch "master", while GitHub has renamed it to "main".



- In **feature branching**, you branch off and work on a feature, and then merge back to the main branch.

HEAD -> master



- HEAD is simply a pointer that refers to the **current location** in your repo. It points to a particular **branch reference**. When you change HEAD, you are just changing to a different branch's reference point.
- In the picture above, there are 2 branches: Master and DarkMode. Each branch has a **branch reference point**, ie Master branch reference point is on the second commit while DarkMode branch reference point is on the fourth commit. When you switch HEAD around, you are changing between the reference points, ie switching between the 2nd and 4th commits.
- It is not true that HEAD *always* points to the last commit on a branch – there is a detached HEAD state to be covered later.
- Protip: a "branch" is really just a pointer to a commit. When you delete a branch, **the commits do not disappear**, only the pointer is removed. Think of it as removing the sticky note on the commit. In other words, commits are NOT tied to a specific branch; one commit can be part of multiple branches' histories.
- Protip: one might ask why "when creating new branches it would not cause a new folder to be created locally, so each branch = items of that branch's content". This is because **a branch is just pointing to some commit, and that commit is associated with specific files and their contents**. Git stores compressed versions of every file in your repo of the .git folder. For eg, if you have three versions of a file across three branches, Git is storing compressed versions of each behind the scenes; you just cannot easily read or make sense of them.

How HEAD Actually Works

- If you *cd* into your .git folder and *cat HEAD*, you will realise HEAD is a single string of "ref: refs/heads/x", where x is where your current HEAD is pointing to. Hence, HEAD is just referencing a branch.
- OTOH, the "refs/heads/x" is referencing a particular commit. If you type *ls* you will realise there is a folder path of refs > heads which lead to various files of all your branches. If you open each file, you will realise it just contains 1 commit hash of the commit the branch is pointing to currently.

Switching When Uncommitted Work

- When you switch to a branch without staging / committing changes you have made in your current branch, you will run into an error where it asks you to commit or stash these changes before changing branches. If not, you will lose those changes. **This error only happens when two branches have that file and they are in conflict with each other (content is different**.
- However if one branch had an EXTRA file that none of the other branches have, you can *git switch* without error and that extra file will come with you. **Unstaged files follow you.**

2.2 Branching Commands

- View existing branches (* is current branch): **git branch**
- View with more info: **git branch -v**

- Create new branch **based on the current HEAD** (so where you are currently is important because your branch starts from there). Merely creates the branch and DOES NOT switch you to that branch: **git branch <branch-name>**. We now have 2 branch references pointing to the same commit.
- Switch HEAD to another branch: **git switch <branch-name>**. This is a newer command whereas the older command is **git checkout**.
- Switch to previous branch because you forgot: **git switch -**

Alternative Switching

- Checkout command but it does a million additional things so the decision was made to add a standalone switch command which is much simpler: **git checkout <branch-name>**. The additional things is to "restore working tree files" (see docs).
- Create branch and then switch to it: **git switch -c <branch-name>** where -c means create.
- Another create branch and switch: **git checkout -b <branch-name>**

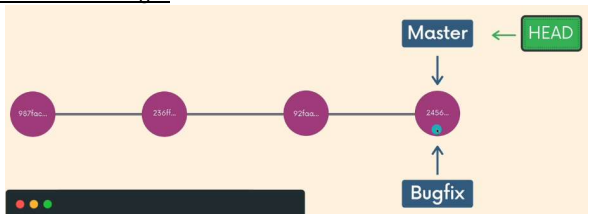
Renaming and Deleting

- Delete, but (i) you cannot be at that branch to be deleted, and (ii) that branch must be fully merged: **git branch -d <branch-name>**
- Delete, ignoring fully merged status: **git branch -D <branch-name>**
- Rename **YOU MUST BE AT THE BRANCH YOU WANT TO RENAME FIRST** (m for move): **git branch -m <new-branch-name>**
- **What is fully merged?** It means you can use -d to delete a branch only if its changes have been merged into some other branch, ie the other branch must contain 100% of all content of the branch you want to delete. In other words, the branch you want to delete must be a subset of another existing branch.

2.3 Merging

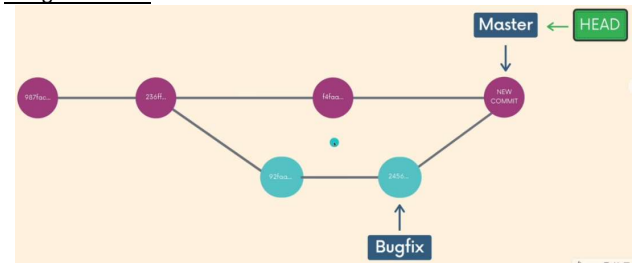
- ★ Merging can be confusing, so always remember these two concepts:
  - **We merge branches, not specific commits. We do not pick specific commits to merge.**
  - **We always merge to the current HEAD branch, ie we merge to where we are.**
    - **Switch to the branch you want to merge the changes into (the receiving branch).**
    - **Use *git merge <branch-name>* to merge changes from a specific branch into the current branch.**

Fast Forward Merges



- A **fast-forward merge** is the simplest where you simply move the master to a later commit. This is only when there are no extra commits on master in relation to bugfix, ie master is a subset of bugfix. The master was pointing the second commit, and doing *git merge bugfix* will just move the master to the commit of the bugfix pointer. **It is basically just catching up a laggard pointer to the newer one.**
- After merging, the branches are still separate and are NOT forever in sync.

Merge Commits



- This is for automatic merges done by Git and assumes no conflicting files (if a file add extra lines = no conflict, but if a line has different words between two files = conflict).
- Assume master is at 3rd purple commit while bugfix is at 2nd blue commit. Git will then generate a **merge commit**, which is just Git making a commit on the recipient branch. This new merge commit now has two parent commits.
- You do the same steps as above. This time, your editor will open up for comments. Close editor to make the commit. You will see "merge made by recursive strategy".

Merge Conflicts

- This happens because of conflicts and Git does not know how to handle these conflicts. You must manually fix them.
- Your files now have new things that indicate which came from which file. Content from your current HEAD is displayed above while content from branch you are trying to bring into current is below.
- Ask yourself: do you want to keep them as separate branches? Maybe merge into a third branch?
- Steps:
  - git switch <recipient-branch>
  - git merge <other-branch>
  - Fix conflicts.
  - git add file.txt
  - git commit -m "resolve conflicts"
  - DONE!

Chapter 3: Git Diff

3.1 Understanding Git Diff

- **Git Diff** is all about showing changes between commits, branches, files, our working directory, and more! Often used with *git status* and *git log* to get a better picture of the repo.

Git Diff

- **git diff** (without additional options) lists all the changes in our working directory that are NOT staged for the next commit, ie **compares files in the staging area with what they are in the working directory**. AKA it is **showing us unstaged changes**.

```
diff --git a/rainbow.txt b/rainbow.txt
index 72d1d5a..f2c8117 100644
--- a/rainbow.txt
+++ b/rainbow.txt
@@ -3,4 +3,5 @@ orange
 yellow
 green
 blue
-purple
+indigo
+violet
```

- This tells you:
  - First line tells you the files actually compared. One file is called "a" and other called "b". Usually same file over time.
  - Second line you usually ignore. Each file gets its own hash, while the third number is some internal file mode identifier.
  - Next two lines show you the files again.
  - Next few lines show you the actual change, and some lines before and after for context.
  - The @@ -3, 4 +3, 5 @@ means that:
    - Minus sign for file "a", as indicated in third line.
    - 3,4 means 4 lines extracted starting from line 3 for file "a". 3, 5 means 5 lines starting from line 3 for file "b".
    - Anyway not very important.

- **git diff HEAD** lists all changes in the working tree since your last commit, AKA **show all changes both staged and unstaged**. Use it to see what changes you have made so far but not committed.
- **git diff HEAD~123** or **git diff commit-hash-no** to compare HEAD and what you typed.
- **git diff --staged** OR **git diff --cached** are both the SAME commands that list the changes between the staging area and our last commit (so only staged changes). Another way of seeing it is **"show me what will be included in my commit if I run git commit now"**.

Specific Files

- The above do all files. If you want specific files then do the below.
- **git diff HEAD [filename] [filename2]**
- **git diff --staged [filename] [filename2]**

Comparing Branches

- **git diff branch1..branch2**
- The order of the branches matters as the output will be different. Must know how to read. You are comparing branch1 to branch2 now.
- You can do a space instead of dot dot. Both works well.

Comparing Commits

- **git diff commit1..commit2**

Chapter 4: Stashing

4.1 Intro to Stashing

- Note that stashing is lesser used, but if you do use it, mainly just *git stash* and *git stash pop*. The rest are very niche cases.
- Stashing solves a problem where when you are working and you are not done so you do not want to make any commits, but yet need to switch to a different branch.
- Normally when you switch branches and you have uncommitted work, there are two possible outcomes: (1) those changes come with you to the destination branch (because there are no conflicts), or (2) Git will not allow you to switch if it detects potential conflicts.
- Git Stash helps to pause and save your changes without actually committing them: **git stash** OR **git stash save** (both same). It will take all uncommitted changes (staged and unstaged) and stash them, reverting the changes in your working copy.
- Remove most recently stashed changes in your stash and reapply them to your working copy: **git stash pop**. Note that you can reapply them to a different branch or same branch that you stashed them on.
- Use **git stash apply** to apply whatever is stashed away, without removing it from the stash. This is useful if you want to apply stashed changes to multiple branches.
  - Note that you might face a conflict error message. Just choose what you want.

Multiple Stashes

- Can keep typing *git stash* to add multiple stashes. They will be stashed in the order you added them (LIFO).
- See your stash with **git stash list** where **@{0}** is the topmost to be popped (and last stashed). The @{number} is the ID of the stash.
- If you want a specific stashed save instead of popping the top, you can reference the ID with **git stash apply stash@{2}**

Dropping and Clearing Stashes

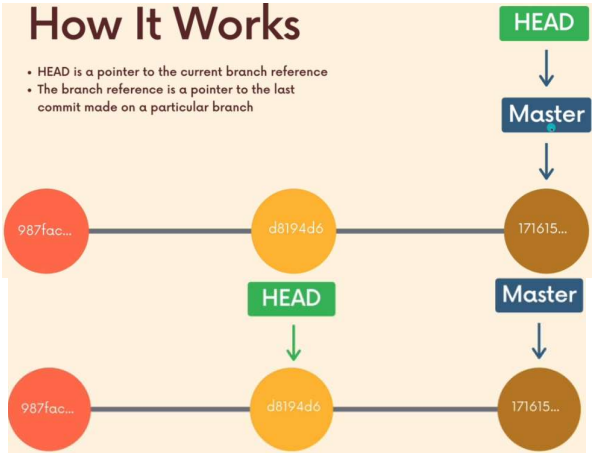
- Drop specific: **git stash drop stash@{2}**
- Drop all: **git stash clear**

Chapter 5: Undoing Changes and Time Travelling

5.1 Intro

Git Checkout

- Many developers think that *git checkout* is overloaded (does too many things), which is why *git switch* and *git restore* were created. Git checkout can create branches, switch to new branches, restore files, and undo history!
- See a specific commit: **git checkout <hash>** (just need first 7 digits)
  - When you run this command, you will see a scary looking message: "You are in 'detached HEAD' state…".
  - If you type *git log –oneline*, you only see the commits up to that point. This is because you have jumped back in time to a previous commit.
  - **So what is detached HEAD**? Usually, HEAD points to a specific branch reference rather than a particular commit. But when you checkout a particular commit, HEAD points at that commit rather than at the branch pointer. DO NOT PANIC. Having a detached HEAD means you cannot work on the files, only view.
  - If you cat .git/HEAD, you will see the commit hash instead of things like refs/heads/branchname.
- What to do with detached HEAD:
  - Stay in detached HEAD to examine the contents of the old commit. Poke around, view files etc.
  - Leave and go back to wherever you were before by *git switch branchname*. This is also called **reattaching the HEAD**.
  - Create a new branch and switch to it. You can now make and save changes, since HEAD is no longer detached.

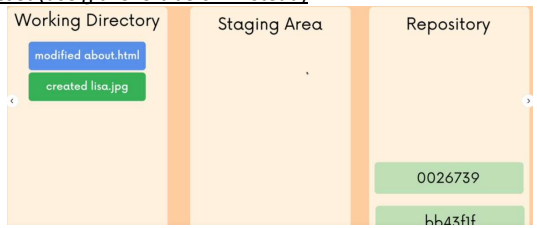


- You can use *git checkout* to revert the file back to whatever it looked like when you last committed, aka to discard any changes since the commit: **git checkout HEAD <file1> <file2>** OR **git checkout -- <file1> <file2>** (same commands)

Git Restore

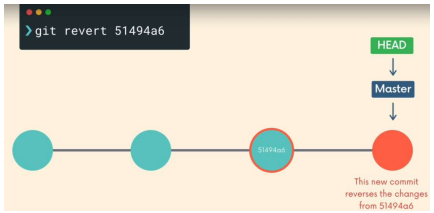
- This is a brand new command that helps with undoing operations. Because it is so new, most tutorials do not mention it but it is worth knowing.
- While *git checkout* does many things and so is confusing, **git restore** was introduced alongside *git switch* as alternatives to some of the uses for *checkout*.
- To restore file to contents of the commit wherever HEAD is (discard changes since last commit): **git restore <file1>**. Same as *git checkout HEAD <file1>*.
- To restore file to contents of a specific commit: **git restore --source HEAD~1_or_hash <file1>**
- To remove staged files: **git restore --staged <file1>**

Git Reset (use git revert below instead)



- This is to undo commits on a branch. It will reset the repo back to a specific commit; the commits will be GONE. However this is a soft reset and THE CHANGES WILL STILL BE THERE IN YOUR WORKING DIRECTORY – GIT JUST DELETES THE COMMITS BUT DOES NOT REVERSE ANY CONTENT. This is useful if you made some commits on the wrong branch but you want to still keep that work for a future branch: **git reset <hash>** then possibly *git switch -c newbranch* and then *git commit -am "to new branch"*.
- To undo both commits and the actual changes in your files: **git reset --hard <hash>**

Git Revert



- Git Revert is very similar to Git Reset in that they both "undo" changes, but in different ways. Git Reset moves the branch pointer backwards, eliminating commits. Git Revert instead creates a brand new commit which reverses / undoes the changes from a commit, and because it results in a new commit, you will be prompted to enter a commit message: **git revert <hash>**

- If you want to reverse some commits when collaborating with others, use revert. If you want to reverse commits that you do not want others to know, use reset.

5.2 Summary

- Go to specific commit: **git checkout <hash>**
- Revert to last commit (discard changes): **git checkout HEAD <file1>**
- Revert to last commit (discard changes) **git restore <file1>**
- Revert to specific commit: **git restore –source <hash> <file1>**
- Unstage files: **git restore --staged <file1>**
- Revert to specific commit (no discard of changes) and delete commits beyond: **git reset <hash>**
- Revert to specific commit (discard changes) and delete commits beyond: **git reset --hard <hash>**
- Revert to specific commit (discard changes) but by creating a new commit: **git revert <hash>**

Chapter 6: GitHub

6.1 Introduction to GH

- GitHub is a hosting platform for git repos. It also provides additional collaboration features that are not native to Git but are super helpful.

Cloning

- Get a local copy of an existing repo (make sure you are NOT inside of a repo when you clone!): **git clone <url>**

Git Remote

- Before you can push anything up to GH, you need to tell Git about our remote repository on GH, ie the destination to push up to. In Git, we refer to these destinations as **remotes**, which is simply a URL (and its label) where a hosted repo lives.
- View existing remotes for your repo: **git remote -v** (-v optional)
- Add new remote: **git remote add <name> <url>** (a very standard name is "origin". So every time you are calling for "origin", you are telling Git to call this URL.)
- Rename remotes: **git remote rename <old> <new>**
- Delete remote: **git remote remove <name>**

Uploading

- Pushing up (you are only pushing the branch you type in): **git push <remote> <branch>**, usually ***git push origin master_main***.
  - Note how you DO NOT push up a particular commit, but instead the entire branch.
  - You also do not need to be on that certain branch before you can push to it. GH will match the branch name with the one in the repo; if not exists, will create one.
- While most of the time you want to push a local branch up to a remote branch of the same name, you do not have to! This pushes local pancake branch up to a remote waffle branch: **git push origin pancake:waffle**, or **git push <remote> <local-br>:<remote-br>**. This is a rare use case but showcases that the branches do not have to be named the same.
- The **-u option** allows us to set the upstream of the branch we are pushing. Think of it as telling your computer to remember a link connecting our local branch to a branch on GH. Running **git push -u origin master** sets the upstream of the local master branch so that it tracks the master branch on the origin repo. This allows you to just type *git push* (that's all!).

Renaming Branches

- Renames to main: **git branch -M main**

6.2 Getting Started with GH

Steps:

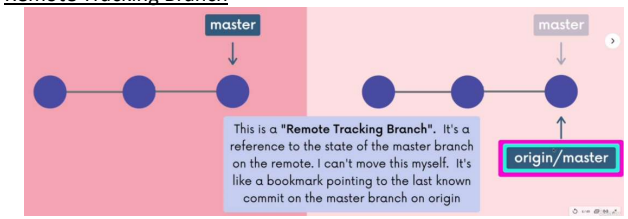
- Register and sign in an account at GH.
- Set up your SSH keys so that you are authenticated to do certain stuff. If you do not set up, then you will be prompted for Email and PW every single time you use GH.
  - Type *ls -al ~/.ssh* to list your existing SSH keys.
  - IF NOT, generate a new one. Google how to do so and follow the instructions (easy to follow).

Uploading Stuff into GH

- There are two methods.
- Method #1 is when you have an existing repo locally:
  - Create a new repo on GH ➔ + sign > Create new repo > Fill up details.
  - Connect your local repo (add a remote).
    - Connect (**use the SSH version**): *git remote add origin <url>*
    - git remote -v # see if worked
    - If you accidentally used the HTTPS version and want to change to SSH, do: *git remote set-url origin <url>*.
  - Push up your changes to GH: *git push origin master*
  - Protip: usually what you would do is locally merge your feature branches into master/main, then push master/main up into GitHub.
- Method #2 is when you have not started any work yet:
  - Create a brand new repo on GH.
  - Clone it down to your machine.
  - Do some work locally.
  - Push up your changes to GH.

6.3 Fetching & Pulling

Remote Tracking Branch



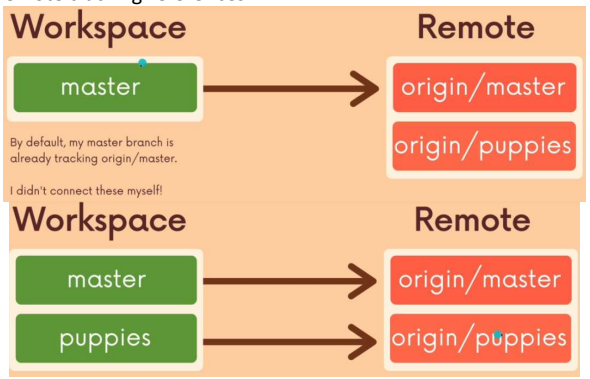
- When you clone a GH repo onto your local computer, you copy paste all commits. There are two pointers – one is the top "master" one that is the same as the GH repo's, and another is called the **remote tracking branch reference**. This second pointer is not moved by the user; it points to the last known commit on the master branch on the origin remote. Hence as you do more work locally, the top pointer moves but the bottom one stays; bottom only moves when you communicate with the GH repo again and it gets updated.
- Its name in the picture is "origin/master", but basically it follows the pattern of **<remote>/<branch>**. For eg:
  - Where **origin/master** references the state of the master branch on the remote repo named origin
  - Or **upstream/logoRedesignetc** references the state of the logoRedesignetc branch on the remote named upstream (a common remote name).
- View remote branches: **git branch -r**
- You will also see an **origin/HEAD** which is a pointer to a specific branch / commit that will be checked out when a user clones the repo.

Additional Behaviours

- The above behaviour of the two pointers is useful because you can then use *git diff* to figure out the differences between the local copy and the remote copy.
- Can also type *git status* to see how far ahead you are.
- You can *git checkout origin/main* to see.

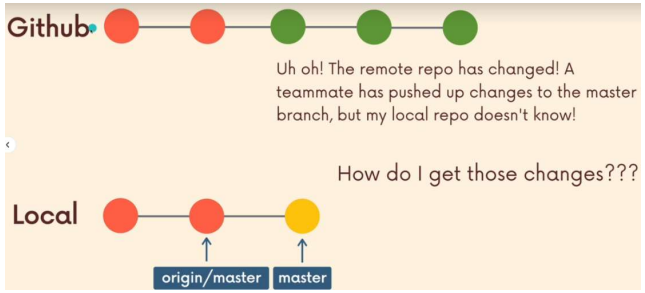
Working with Remote Branches

- When you *git clone* a repo, and you type *git branch* in it, it only shows the main/master branch and not the other branches. What you must do instead is type *git branch -r* to show you all those remote tracking references.

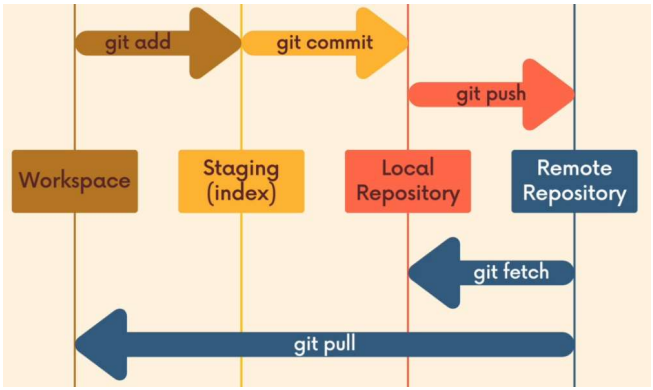


- When you clone a repo, the local master branch will be automatically tracking the remote origin's master branch. But what if you want to work on the other remote branches?
  - You could *git checkout origin/puppies*, but that puts you in detached HEAD. But this is not what you want, because you cannot work on your files with a detached HEAD as it is pointing to a floating commit that is not rooted to any local branch. What you want is to have your own local branch called *puppies* that is connected to *origin/puppies*.
  - What you want to do is type **git switch <remote-branch-name>** such as *git switch puppies*. This will make a local branch from the remote branch of the same name AND set it up to track the remote branch.
  - Note that the old command used to be more complicated: **git checkout --track origin/puppies**

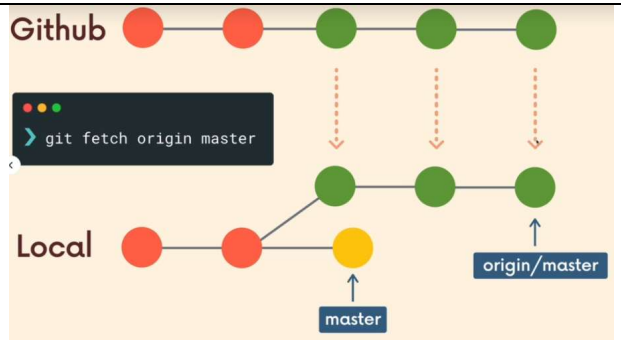
Git Fetch



- When collaborating with others, the main would have changed while you were working locally. This is where git fetch and pull come in handy.



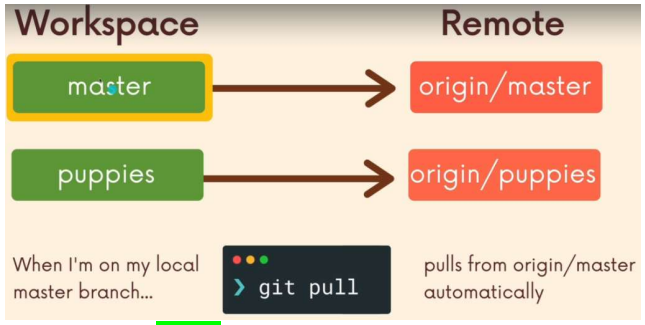
- **Fetching** allows us to download changes from a remote repo BUT those changes will not be automatically integrated into our working files. It lets you see what others have been working on without having to merge those changes into your local repo.
  - Fetch all branches: **git fetch <remote>** (if not specified, <remote> defaults to origin)
  - For eg, *git fetch origin* fetches all changes from the origin remote repo.
  - Fetch specific branch: **git fetch <remote> <branch>**, for eg *git fetch origin master* retrieves the latest info from the master branch on the origin remote repo.
  - **FOR BOTH:** you will not see anything in your VSC. You must first ***git checkout origin {branch}*** and then look at VSC.
- **WARNING**: If you fetch a remote branch and decide to make changes to it, you should NOT work directly on that remote-tracking branch because of potential obstacles. Instead, you should merge those changes to the local branch and then work on them. Otherwise, just use *git pull*.



Notice how the origin/master pointer (remote reference pointer) will be updated. You now have those changes on your machine but if you want to see them need to "checkout origin/master", leaving your master branch untouched.

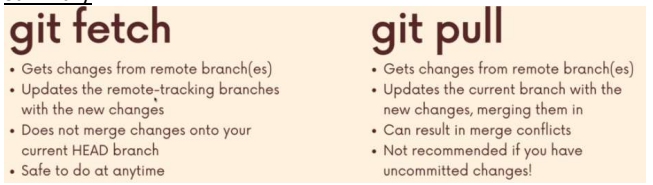
Git Pull

- Unlike fetch, **git pull** actually updates our HEAD branch with whatever changes were retrieved from the remote. Basically it is a fetch + merge.
- It matters WHERE you run the command from – whatever branch you run it from is where the changes will be merged into.
- Pulling: **git pull <remote> <branch>**, for eg *git pull origin master* will fetch the latest info from origin's master branch and merge those changes into our current branch.
- As all merges go, there may be merge conflicts. Solve them, then when done you can *git push origin branchname*.



- If you just run **git pull** without specifying anything else, git assumes the following: (i) remote default is origin, (ii) branch will default to whatever **tracking connection** is configured for your current branch. A tracking connection is that connection from workspace connecting to the remote (see picture under "Working with Remote Branches").

Summary



6.4 More with GH

Public vs Private Repos

- Public repos is visible and clone-able by anyone. Private repos can only be seen by collaborators you give permissions to; for enterprise accounts, a private repo may be visible to everyone within the organisation (need to grant permission).

READMEs

- Used to communicate: (i) what project does, (ii) how to run it, (iii) why it is noteworthy, and (iv) who maintains the project.
- If you put it in the root of your project, GH will recognise it and auto display it on the repo's home page.
- Written in **Markdown**, a convenient syntax to generate formatted text.

Collaboration Process

- Simple Procedure (O = owner, C = collaborator):
  - C: Git clone to your machine and work on it.
  - C: Git add and commit and push to the remote repo.
  - O wants the new stuff. Assuming fast-forward merge (no conflicts), O will just *git pull origin main* so he will have the updated files locally. The end.

GitHub Gists

- **GH Gists** are a simple way to share code snippets and useful fragments with others. Very similar to PasteBin. Gists are much easier to create, but offer far fewer features than a typical GH repo.
- Profile > Your gists.

GH Pages

- These are public webpages that are hosted and published via GH, allowing you to create a website simply by pushing your code to GH.
- Only for static webpages (HTML/CSS/JS). Does not support server-side code.
- There are two types of sites: **User Site** (one per GH acct), usually for your own portfolio with address of *username.github.io*; or **Project Sites** (unlimited), where each GH repo can have a corresponding hosted website by telling GH which specific branch contains the web content, usually *username.github.io/repo-name*.
- Steps:

- Create a remote repo on GH.
- Git init a project locally. Add in content. Rename to main. Add the remote address, and then push content up.
- Select a branch that has the webpage info. That branch must have a file called **index.html**.
- Settings > scroll down until GitHub Pages > choose source (the branch) and the folder containing the index.html file.
- The end!

Chapter 7: Git Collaboration Workflows

Take note that the names of these workflows are not real, just what the author uses to describe.

7.1 Centralised Workflow

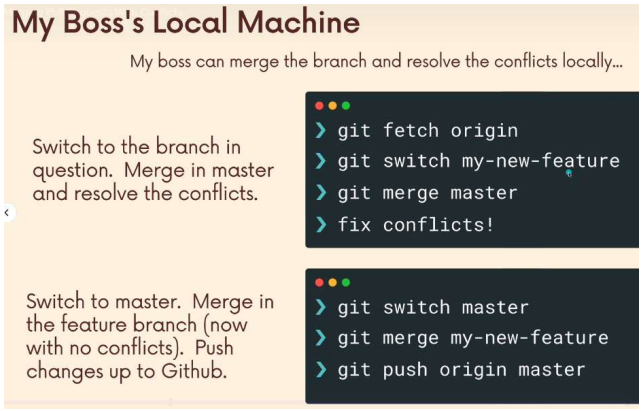
- This is the most basic workflow possible where **everyone works on the master/main branch**. It is straightforward and can work for tiny teams.
- Each user has to *git pull* the most updated work on the remote repo, resolve conflicts, and then *git push* it back.
- Cons:
  - Lots of time spent resolving conflicts and merging code, especially as team grows bigger.
  - No one can work on anything without disturbing the main codebase.
  - Only way to collaborate on a feature together with another teammate is to push incomplete code to master. Other teammates then have to pull it down and now have broken code…

7.2 Feature Branch Workflow

- Unlike above, **no one ever works on master branch, all new development should be done on separate feature branches**. This means:
  - Treat master/main as the official project history.
  - Multiple teammates can collaborate on a single feature and share code back and forth without polluting the master/main branch.
  - Master/main branch will not (hopefully) contain broken code.
- As a user:
  - Everyone clones from master using *git pull origin main* to have the most updated master, then works on their own branches.
  - Do *git fetch origin* to see what branches are in remote repo. You can briefly check out their work with *git branch -r* and then *git checkout origin/navbarx* (detached HEAD so just view). When done looking, reattach HEAD with *git switch -*. Then can *git switch navbar* where Git will automatically connect for you the branches. ➔ **Q: why need this checkout thing? Why not immediately *git switch featurebranch*?** This is because checking out origin/feat allows us to look at the state of the branch on the remote repo (most updated); if just switch to feat only, it is local, and may not be updated.
  - ALTERNATIVELY, do *git pull origin navbar* to pull others' branches into local. This is so you can check their work. Note how this does not interfere with their master nor the current branch they are working on. ➔ **Q: This method better or the above?** Both are personal preference. The former allows you to just see the code because maybe you are not planning to make any changes so you can checkout and let your colleague know what to fix. But only if it is complex issue would you want to save it locally.
  - After checking and editing their colleague's work, they can *git add and commit*, then *git push origin branchname* their colleague's branch back to the remote repo so their colleagues can see it.
  - If you are happy with your work, can first check your local copy of master is updated by first switching back to master *git switch master*, then *git pull origin main*, then when done *git merge feature* into master. (Or rather, you follow the series of checks and approvals / PRs etc. More to be discussed in future). Finally *git push origin main*.
  - If instead you are the colleague having problems, then locally add and commit your work on your feature branch, and then *git remote -v* to check your remote repo name, then *git push origin yourbranchname*.

7.3 Merging in Feature Branches

- There are a couple of ways to merge feature branches:
  - Merge anytime you want, without any sort of discussing with teammates.
  - Send an email / message to discuss if changes should be merged.
  - **Pull requests**!
- **Pull requests** are a feature built into products like GH. They are NOT native to Git itself. They allow developers to alert team members to new work that needs to be reviewed. They provide a mechanism to approve or reject the work on a given branch. They also help facilitate discussion and feedback on the specified commits.
- The workflow would be:
  - Do work locally on a feature branch.
  - Push up feature branch to GH.
  - Open a PR using the feature branch just pushed up.
  - Wait for the PR to be approved and merged.
- What if there are conflicts in the PR?
  - Method 1: Resolve conflicts > edit in the browser.
  - Method 2: follow commands in the "view command line instructions". Locally (for person that reviews PR):
    - View changes locally: *git fetch origin*
    - Switch to that branch: *git switch new-heading* (assuming the remote branch is origin/new-heading).
    - Bring main over to this feature branch, where you can resolve conflicts: *git switch main* (to switch and pull latest main content) > *git pull origin main* > *git switch new-heading* > *git merge main* > manual resolve.
    - Add and commit.
    - Switch back to main to push: *git switch main* > *git merge --no-ff new-heading* ("no ff" because a fast forward unlike merge will just move branch pointer to some new commit, which is not what we want if we want to preserve and make it clear there was a branch merge in our history; optional to do it is just what GH recommends) > *git push origin main*.
    - GH will let you know the successful merged PR which is closed now. Feel free to delete the feature branch.



Configuring Branch Protection Rules

- Configure many things at: Settings > Branches
- Set up a **branch name pattern** so branches follow a certain pattern.
- Can **require PRs before merging**, disallowing people to merge directly. Someone else must review and then accept.

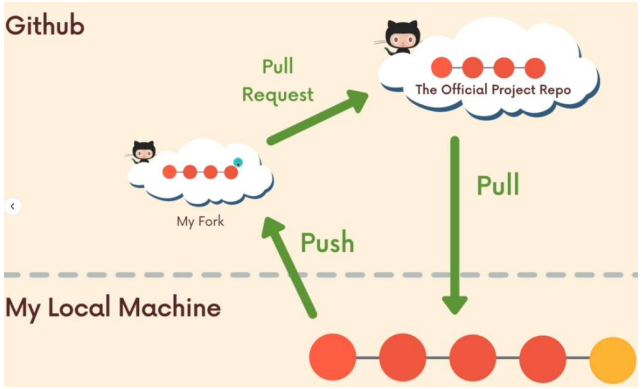
7.4 Forking and Cloning

- The **fork & clone workflow** is different – instead of one centralised GH repo, every developer has their own GH repo in addition to the "main" repo. Developers make changes and push to their own forks before making PRs.
- Most common in large open-sourced projects where they may be thousands of contributors with only a couple maintainers. No need to invite thousands of people as contributors.
- GH (and similar tools) allow us to create personal copies of other peoples' repos on our GH accounts. We call those copies a **fork** of the original. As with PRs, forking is not a Git feature – it is implemented by GH.

- **Q: why not just clone?** This is because if you clone and make changes, you cannot just push up as a PR unless you are given permission as a collaborator.

Forking Workflows

- Fork from original repo. A new repo is created in your GH account.
- Clone from your repo. There will be a default remote name of *origin* that is created automatically when you cloned.
- Set up a second remote (normally called *upstream* or *original*) which refers to the ORIGINAL REPO (the one you forked from). This system allows you to push changes to your own forked repo, and also pull down any new changes from the original repo. This is done via: **git remote add upstream <original repo's address in https(?)>** (where upstream is the name you are giving it). Type *git remote -v* to see the new additions. For the actual pull down of changes, do: *git pull upstream main* (or whatever branch).
- Make changes, add, commit, push to your own forked repo *git push origin main*.
- Want to merge to the original repo? Just create a PR from the repo page. If approved, great!
- On a new day, pull down changes from the upstream to keep your local files updated. Any new changes will be pushed to your forked repo. Repeat.

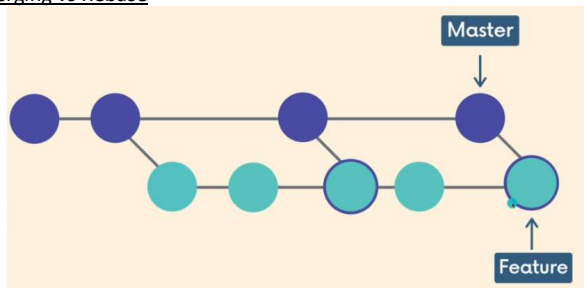


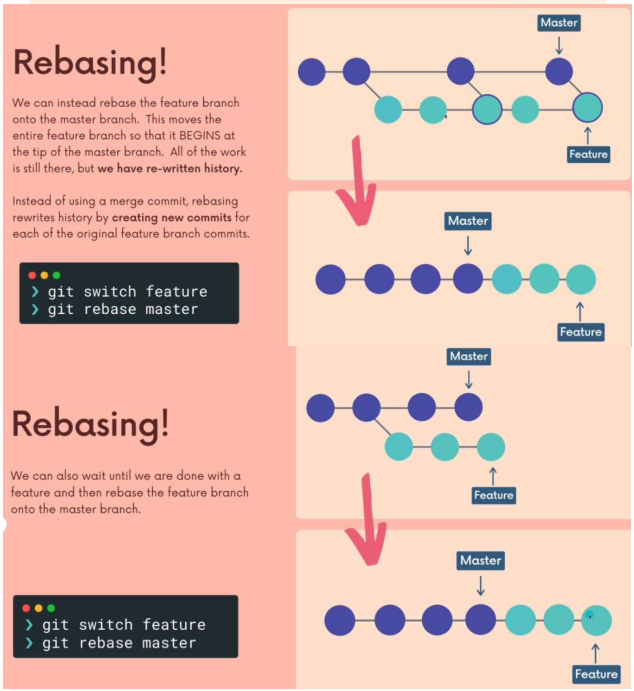
Chapter 8: Git Rebase

8.1 Introduction to Rebase

- Not the most popular, sometimes even actively avoided. Yet it is increasingly used in engineering companies.
- **Rebasing** has two uses: (1) as an alternative to merging (git rebase instead of git merge), or (2) as a cleanup tool to clean up your commits and your git history.

Merging vs Rebase





- In git merge workflows, you will work on a feature branch. The key point is that you must also check regularly the remote repo to see if the master branch was updated or not, and if so, to merge down to your local copy. This results in a new merge commit (green dot with purple outline).
- The feature branch has a bunch of merge commits. Over time, if your master branch is very active, your feature branch's history when you git log will be very muddied and cluttered as well.
- OTOH, when you **rebase** the feature branch onto the master branch, it moves the entire feature branch so that it BEGINS at the tip of the master branch. All of the work is still there, but we have rewritten history. Instead of using merge commit, **rebasing rewrites history by creating new commits for each of the original feature branch commits**. ➔ Overall, you are changing the base by making the master branch your new base. You end up with a linear structure, where the feature branch is added at the tip of the master branch.
- Steps:
  - You want to be on your feature branch first: **git switch feature**
  - Then call: **git rebase master** (note that this master is your local master branch, not the remote one. So you may need to git pull your master and make sure it is the most updated version)
  - You can choose to (i) merge then rebase, or (ii) rebase right at the start without merging (same commands). Both same results.
- Note the mechanism: rebase goes through all the commits on the feature branch and recreates each one right at the tip of a branch.

8.2 When Not To Rebase

- Rebasing rewrites the history of commits. Hence, never rebase commits that have been shared with others. If you have already pushed commits up to GH, DO NOT rebase them unless you are positive no one on the team is using those commits. You do not want to rewrite any git history that other people already have. It is a pain to reconcile the alternate histories!
- In other words, you should only rebase commits that you have on your machine that other people do not, ie your feature branches you are working on. This is because others have that master/main branch and have those commits already.
- When you rebase your work from your feature branch ONTO the master branch, you are not changing the master branch, you are not changing the branch you are rebasing on. **Only the feature branch you are on when you rebase is the one having its commits replaced**. This is okay.
- CONCLUSION: Never rebase any commits that others have. You can only rebase stuff that is yours and yours alone.

Others

- Avoid rebasing if your feature branch has sub-branches of its own. This is because the sub-branches will be "loose" without a commit to refer to (because the original commits are recalculated).

Managing Conflicts

- When running into conflicts, you can choose to abort it, or you can do so like usual, but then run *git add <conflicted-file>* and *git rebase --continue*.

8.3 Interactive Rebase (rebase as a cleanup tool)

- Git Rebase allows us to rewrite, delete, rename, or even reorder commits (before sharing them). This is useful when your commits have terrible messages, buggy features, or you just want to streamline multiple commits into a combined few.
- You do so with the **interactive -i** option which allows us to edit commits, add files, drop commits etc. Need to specify how far back we want to rewrite commits: **git rebase -i HEAD~4** (here we are specifying a range of 4 commits to go back and recreate each one)
- Also note that we are NOT rebasing onto another branch. Instead we are rebasing a series of commits onto the HEAD they currently are based on.

Example

The initial commit log:

```
2029e20 my cat made this commit
655204d fix another navbar typo
2a45e71 fix navbar typos
4ff2290 add top navbar
6e39a76 whoops forgot to add bootstrap js script
240827f add bootstrap
519aab6 add basic HTML boilerplate
cbee26b I added project files
0e19c7a initial commit
```

The end goal:

```
b8f8889 add top navbar
64fa7d4 add bootstrap
4273423 add basic HTML boilerplate
0ffb0aa add project files
0e19c7a initial commit
```

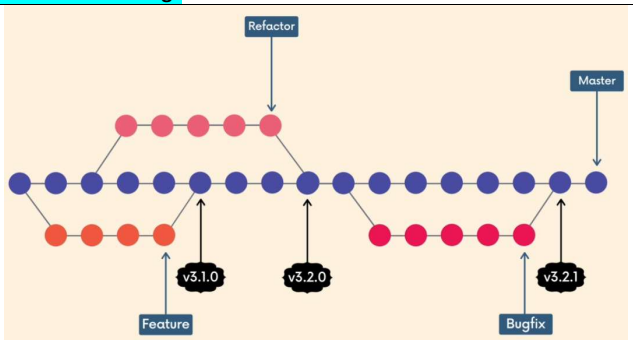
- https://github.com/Colt/interactive-rebase-demo (add one more Create README.md commit at the top)
- Create feature branch: **git switch -c my-feat**
- From the README commit to the 'added' commit, there are 9 commits to jump to (1 starts from topmost commit). Then you want to go back: **git rebase -i HEAD~9**
- Opens text editor w/ a list of commits (reversed order) alongside a list of commands to choose from. The order is how they will deal with each commit starting from the top. The more commonly used commands are:
  - **pick** = use the commit (AKA no change)
  - **reword** = use the commit, but edit the commit message
  - **edit** = use commit, but stop for amending

- **fixup** = use commit contents but meld it into previous commit and discard the commit message
- **drop** = remove commit

- Our first commit is to reword it. So we change "pick" to "reword", save file, close the VSC window for the commands, a new window pops up allowing you to edit to the commit message you want. Our example is "I added project files" to "add project files" verbatim. The end. If you *git log*, notice how all your commits have changed hash because they were rewritten.
- For our second commit, we can start with *git rebase -i HEAD~9* (9 is not necessary but why not). Our next thing to edit is the "whoops forgot" line which we will use *fixup* instead. It takes the commit from before and adds the fixup-ed one, and throws away the fixup-ed's commit message. The end.
- The next one is to combine the navbar ones. Remember you have one fewer commit so do *git rebase -i HEAD~8*. For the "fix navbar typos" and "fix another navbar typos", we change both to *fixup* because then they will merge to the previous commit and throw away their messages, effectively combining them.
- Lastly, need to remove a commit which ALSO REMOVES THE CHANGES. We need *git rebase -i HEAD~2*, then change the "cat" one to *drop*. The end.

Chapter 9: Git Tags

9.1 Intro to Git Tags



- Basically tags are pointers that refer to particular points in Git history. We can mark a particular moment in time with a tag. Tags are most often used to mark version releases in projects, eg v4.1.0 v4.1.1 etc.
- Think of tags as branch references that DO NOT CHANGE. Once a tag is created, it always refers to the same commit. It is just a label for a commit.
- Two types of tags: **lightweight tags** are lightweight, ie just a name / label that points to a particular commit; and **annotated tags** store extra metadata including the author's name and email, the date, and a tagging message (like a commit message). Projects usually use the latter.

Semantic Versioning

- Tagging is for versioning. The **semantic versioning** spec outlines a standardised versioning system for software releases. It provides a consistent way for developers to give meaning to their releases (eg how big of a change is this release?). Versions consists of three numbers separated by periods, eg 2.4.1.
  - The numbers are major release, minor release, and patch release respectively.
  - The initial first ever release must be 1.0.0.
  - Patch releases normally do not contain new features or significant changes. They typically signify bug fixes and other changes that do not impact how the software is used (how people use your software).
  - Minor releases signify new features or functionalities have been added, but the project is still backwards compatible. No breaking changes. The new functionality is optional and should not force users to rewrite their own code. Remember to reset patch number back to 0.
  - Major releases signify significant changes that is no longer backwards compatible. Features may be removed or changed substantially.
  - You may append things like "beta", eg v3.2.4-beta2.

9.2 Using Tags

Viewing Tags

- Print all tags in current repo: **git tag**
- List (-l) tags that match a pattern: **git tag -l "*beta*"** (notice the wildcards used)

Checking Out Tags

- To view the state of a repo at a particular tag (for a particular commit, NOT A BRANCH), which also puts us in detached HEAD: **git checkout <tag>**
- See difference between two tags: **git diff v17.0.0 v17.0.1**

Creation

- Create lightweight tag. By default, Git creates it referring to the commit that HEAD is at: **git tag <tagname>**
- Create annotated tag. If never use -m option, your text editor will open. If you did, you can pass your message directly: **git tag -a <tagname>**
- View metadata of annotated tag: **git show <tagname>**
- Create tags for older commits (the above do it at HEAD): **git tag -a <tagname> <commit>** (ignore -a if lightweight)

Edits / Deletes

- Force tag (since tags must be unique – you essentially force move it): **git tag -f <tagname> <commit>**
- Delete tag: **git tag -d <tagname>**

Pushing Tags

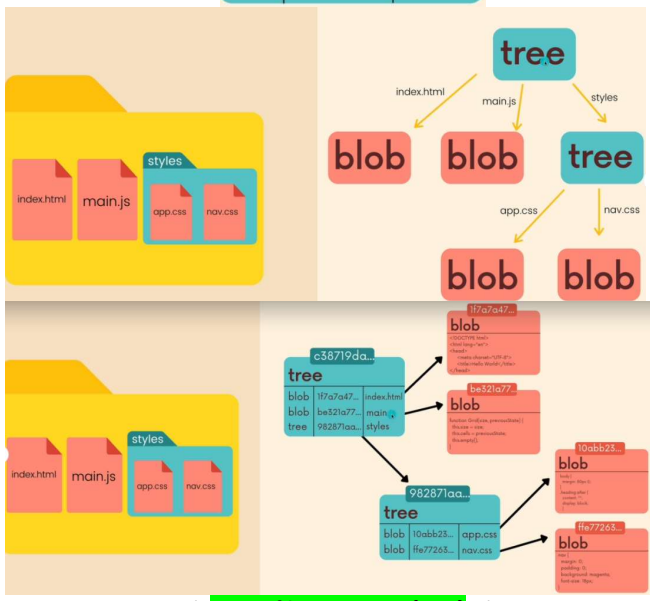
- When you push to GH, tags are not automatically going to be pushed to remote servers. You can use the **--tags option** to transfer all of your tags to the remote server that are not already there: **git push <remote> --tags**
- Push single tag: **git push origin v1.5** (or <remote> <tagname>)

Chapter 10: Git Under the Hood

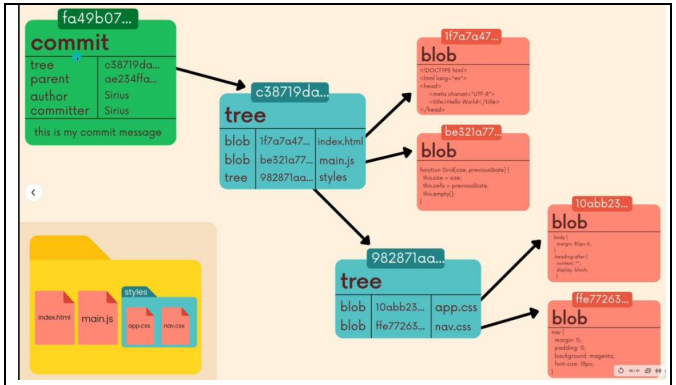
10.1 The .git Folder

The **.git** folder contains (the main ones):

- **objects** folder = the core of Git. It contains all the repo files, like the backups of files, all the commits etc. The files are all compressed and encrypted so they won't look like much. These files are also full snapshots , not diffs or deltas between commits.
  - The 4 types of Git objects: commit, tree, blob, annotated tag.
  - Git is a **key-value data store**. This means any content we put in a repo, Git will return us a unique key to later retrieve that content. These keys that we get back are SHA-1 checksums.
  - The **git hash-object <file>** command takes some data, store in our **.git/objects** directory, and returns us a unique SHA-1 hash. Using just this command alone, Git DOES NOT ACTUALLY STORE ANYTHING – it is the unique key that WOULD be used to store our object.
  - If you do **echo 'hello' | git hash-object --stdin**, you are piping the output of echo to *git hash-object* and *stdin* tells it to use this content from stdin rather than a file. Can write a **-w option** so it will actually be stored. To retrieve them: **git cat-file -p <object-hash>**. Can **git hash-object dogs.txt -w** for files. These commands are all manual stores and are not tied to any commits! Can restore files with **git cat-file -p <hash> > dogs.txt**. ➔ Basically we are understanding how Git stores all these files.
  - All of the stored stuff are **blob (binary large object)** which is the type to store the contents of files. They do not store the filenames of each file or any other data, they just store the contents of a file. They are attached to a blob hash (different hash from commits) where the file names points to it.
  - **Trees** are Git objects used to store the contents of a directory. Each tree contains pointers that can refer to blobs and to other trees. Each entry in a tree contains the SHA-1 hash of a blob or tree, as well as the mode, type, and filename.



- View trees with **git cat-file -p master^{tree}** where master^{tree} specifies the tree object that is pointed to by the tip of our master branch.
- **Commit** objects combine a tree object along with information about the context that led to the current tree. Commits store a reference to parent commit(s), the author, the commiter, and of course the commit message.



- **config** file = for configuration, like how we configure global settings like name and email across all Git repos, but we can also configure things on a per-repo basis (by changing the local config file in your directory).
  - Change name this repo only: **git config --local user.name "hi"**. To see if it worked, run it again without the new name.
  - Local email: **git config --local user.email x@gmail.com**
  - Change colours when you use git:
    - [color]
      - ui = true
    - [color "branch"] # when u git branch
      - local = cyan bold
      - current = yellow bold
    - [color "diff"] # when u git diff
      - old = magenta bold
- **HEAD** file = reference to a branch that points to a commit.
- **index** file
- **refs** folder contains the **heads directory refs/heads** which contains one file per branch in a repo. Each file is named after a branch and contains the hash of the commit at the tip of the branch, eg refs/heads/master contains commit hash of the last commit on the master branch. The refs folder also contains **refs/tags** which contains one file for each tag in the repo. There is also **refs/remotes** that point to your remote branches.

Chapter 11: Git Reflog

11.1 Introduction
• This is when you make mistakes in Git. • • • • f

f