Git and GitHub Udemy Course notes

**Essential Git Functions and Commands**

- Version control system, software that tracks and manages changes to files over time.

* Track changes across multiple files
* Compare version of a project
* ‘time travel’ back to old versions
* Revert to previous versions
* Collaborate and share changes
* Combine changes

- essence of git – create ‘checkpoints’ that you can return to in case you mess up the project.

- other systems, Subversion, CVS and Mercurial

- git vs github (not the same thing)

Git -> don’t need internet to use it, runs local on your machine

GitHub -> hosted in the cloud, makes it easier for collaboration (runs using git)

- git (2 ways of interacting with)

🡪 primarily as a command line tool, used through terminals (expected to be able to use as a software developer)

🡪 can also be accessed using GUI (graphical User Interface), like github desktop,

Easier to use (more visual), but not as fast as using a terminal

Using gitkraken for this course

- Git Bash

🡪 command line interface (Unix-based), required for windows. Windows only have command prompt as their command line interface, which is not Unix-based)

- Setting up git

🡪 Setting up name

*$ git config --global user.name "Sia Kian Zhong"*

🡪 Checking current name

*$ git config user.name*

🡪 Setting up email account

*$ git config –global user.email “kianzhongs@gmail.com*

🡪 checking current email address

*$ git config user.email*

- Basic unix commands (for Git Bash)

🡪 Navigation commands

* List contents of current directory -> <ls>
* List ALL contents of current directory, including hidden files and folders -> <ls -a>
* List contents of specific directory -> <ls (name of target directory)>
* Open file explorer of current directory -> <start .>
* Print path of working (current) directory -> <pwd>
* Change directory to specific-> <cd (path of target directory)>
* Change directory down by 1 folder -> <cd ..>
* Change directory back to main (~) folder -> <cd>

🡪 File and folder creation

* Create a new file -> <touch (path to file name OR multiple file names separated by spaces)>
* Create a new folder/directory -> <mkdir (name of folder/directory OR multiple folder names separated by spaces)>

🡪 File and folder deletion

* Delete a file permanently (not to trash can) -> <rm (name of file OR multiple file names separated by spaces)>
* Delete a folder permanently -> <rm -rf (name of folder/directory OR multiple folder names separately spaces)>

- Git Terminology

🡪 Git Repository -> workspace which tracks and manages files within a folder

- Git Commands (initialising)

* Reports the status of current git repository -> <git status>
* Instantiates/Initialises an empty git repository in current directory (i.e. enables git inside the folder, happens only once at the start of a project) -> <git init>

\*\* Big mistake that beginner Git users make

🡪 Note that all nested folders within a Git repo is tracked by Git, therefore don’t initialise ANOTHER Git repo in a nested Git folder. SEPARATE your Git Repos

🡪 BEFORE <git init> is run, run <git status> to make sure current directory is not a Git repo.

\* can resolve multiple nest git repos by deleted the hidden .git folders

Committing

-> checkpoints in a git repository, along with a message

-> can group new saves in multiple files and folders in 1 commit, typically grouped together for a particular function, e.g. added new features for web application

-> areas in committing

* Working directory
* Staging area
* Git Repository

-> commit messages -> statement that summarises changes made in a commit

Git Commands (committing)

* Adds files onto the staging area to prepare for 1 commit -> <git add (file name)>
* Commit all files in the staging area onto git repository in 1 commit -> <git commit>
* Same as above but with a message (but recommended) -> <git commit -m “my message”>
* Check history of Git Repository -> <git log>
* Add ALL changed files in working directory into staging area -> <git add .>
* Open folder in VS code (once configured) -> <code .>

**Commits in Detail**

Git Documentation (<https://git-scm.com/docs>)

You can refer to the book as well

Atomic commits (i.e. keep each commit focused on a single thing)

* Commits should be focused on a single feature (doesn’t mean 1 file), be a change, deletion or creation of a feature in your project.

Commit message

* Convention is Present-tense imperative style, i.e. like you’re giving orders to the code to perform the tasks
* E.g. Make something do something
* BUT don’t have to follow this, just have to make sure to be consistent and follow the style of the project/company

LONG commit message

* use <git commit> and edit it using VS code or any other configured text editor

Git Logs

* Only shows 1 line of commit message and shortened hash -> <git log --oneline>
* Different features to group commit logs for easy viewing

Amending Commits

* Only works on the LATEST commit
* <git commit --amend>
* 2 scenarios for git commit
  + Missed out a file 🡪 stage file (git add) before running <git commit --amend>
  + Typo/amendment for commit message 🡪 just run <git commit --amend>

Ignoring files

* Certain files in repo that you don’t want git to track
  + API keys, credentials i.e. data that you don’t want to be leaked
  + Operating system files
  + Log files
  + Dependencies and packages (e.g. libraries in Python)
* How?
  + Add a file called .gitignore, inside the file, including the following
    - .DS\_Store (ignore .DS\_Store file)
    - Node\_Module/ (Ignore Node\_Module directory/folder)
    - \*.log (ignore all files with .log extension
* Tools online, gitignore.io
  + Can specify the app that you’re working on, and gitignore.io will provide a starting .gitignore file for you.

**Branching**

What is it? Why use it?

* Projects involve more than 1 person. Multiple people are working on different aspects of the project. Editing one halfway make break the other part, therefore branches are needed to test out different features in parallel.
* Can merge afterwards, in future chapter.

Master branch, or is it main?

* Many developers use master branch as the source of truth, a branch to merge changes into. BUT git sees the master branch as the same as any other branch and will not treat it differently
* Debate on renaming Master branch to Main branch (GitHub already using main, but Git locally still uses master as default branch name)

HEAD

* Refers to the current “location” in your repository, points to a particular branch reference (assign HEAD to the reference of the branch you want to work on)
* Refers to whichever branch is “open”, the one that you’re working on
* HEAD will point to the latest commit hash of the chosen branch

Viewing Branches

* Gives list of current existing branches in repository -> <git branch>
  + The branch that is active has an asterisk next to it
* Gives list with more info on all current existing branches (latest commit msg) -> <git branch -v>

Creating branches

* Creating a new branch based on the current HEAD -> <git branch (branch-name)>
  + Branch name should not include spaces
  + Simply creates branch, doesn’t switch you to the branch (HEAD remains the same)

Switching branches

* Switch to another branch -> <git switch (branch-name)>
  + Once you switch, <git log> will only show the history of the current branch

More branching

* \*\*\*\* More efficient way of adding and committing
  + Add and commit ALL unstaged files & folders -> <git commit -a -m “(message)”>

Another way of Switching?

* Method with more features, can be used to switch branches too -> <git checkout (branch-name)>

Creating and switching

* Creating and switching a new branch in one step -> <git switch -c (branch-name)>
* OR <git checkout -b (branch-name)>

Switch with unstaged changes

* If unstaged changes is in a file/folder common to both branches, Git will prevent switching. -> have to commit or stash before switching
* If unstaged changes is in a file specific to the original branch, the file will follow to the new branch, with the unstaged message in <git status>. Very confusing
* THEREFORE, for now, always commit all changes before switching.

Deleting and renaming branches

* Deleting a branch -> <git branch -d (branch-name)>
  + Have to make sure that you are NOT in the branch you want to delete
  + May have to make sure it is merged properly (Not sure what this means
* Renaming a branch -> <git branch -m (new-branch-name)>
  + Have to make sure you are IN the branch that you want to rename

**Merging**

Why?

* Common workflow in a company is where work is done on a FEATURE branch, instead of the master/main branch. Then the approved work (experimentation) is merged back into the master branch.
* Merge branches, not specific commits
* Always merge to current HEAD branch (i.e. always switch back to master, before merging)
  + <git switch master>,
  + Then <git merge (feature-branch-name)>

\*\*merge branches are not deleted! They still exist. If additional commits are added to feature branch, it will not be on master branch. (only coincide when merge function is called, not forever the same as each other)

Types of merges

* Fast-forward merge -> simplest type of merge.
  + Feature branch has additional commits that master does not have AND feature branch has ALL commits that master branch has.

Merge commits

* Occurs when a fast-forward merge cannot happen, each branch has commits that the other one doesn’t,
* When there is no conflict (I.e. files edited are different on each branch, Git will automatically create a merge commit and request a commit message.
* When there is conflict,
  + Have to manually deconflict. E.g line 77 of file.txt “hi” from master branch but line 77 of file.txt “hello” from feature branch

**Undoing changes and time travelling**

Checking out old commits

* First step of undoing changes, detached HEAD state -> <git checkout (first 7 digits of commit hash)
* Note that HEAD typically just point to a branch by its branch reference (latest commit), and NOT a specific commit. But when checkout is used, HEAD points to the specific commit that you want to change
* A few options for what you can do in this state
  + Check out and read through old files (poking around, view, copy only)
  + Re-attach HEAD to branch reference
  + Create a new branch at detached HEAD and branch off from there

Re-attaching detached HEAD

* To return from detached HEAD state, switch back to any branch, e.g. -> <git switch master>
* To create a new branch and branch off from specific commit -> <git switch -c (new-branch-name)>

Referencing Commits Relative to HEAD

* Reference X commits before HEAD -> <git checkout HEAD~X>
  + Each time you do it, you go back X number of commits
* Go back to whichever commit you were on LAST -> <git switch ->

Discarding Changes with Git Checkout

* Reverting saved files to that of last commit
  + <git checkout HEAD (file names or multiple file names)>
  + <git checkout -- (file name or multiple file names)>

Un-Modifying with Git Restore

* Restore files to LAST commit (uncommitted changes will be discarded)
  + <git restore (file name or multiple file names)>
  + Git status will remind you
* Restore files to a SPECIFIC commit
  + <git restore --source (commit-hash) (filename or multiple file names)
    - E.g. git restore --source HEAD~1 dog.txt
* NOTE: command cannot be reversed, i.e. any uncommitted changes cannot be retrieved once file is restored to any commit point.

Un-Staging Changes with Git Restore

* Un-stage staged files (in a scenario where you stage the wrong files, saves to the file will NOT be undone)
  + <git restore --staged (file name or multiple file names)>
  + Git status will remind you

Undoing Commits with Git Reset

* Git Reset resets the repository to a particular commit, 2 types
  + Regular reset
  + Hard Reset
* ‘Regular’ reset
  + <git reset (commit-hash)>
  + Commits are removed up to specified commit-hash, but changes in files are still intact. This is useful when you committed on the wrong branch but you want to keep the saves.
* ‘Hard’ reset
  + <git reset --hard (commit-hash)>
  + Commits AND changes to files are removed up to specified commit-hash.

Reverting Commits with Git Revert

* Both revert and reset will undo changes to the point of a particular commit
  + Git reset moves pointer backwards, eliminating commits
  + Git revert creates a new commit at a previous commit, having removed changes to files since the specified commit point.
* Git Revert
  + <git revert (commit-hash)>
  + Creates a new commit at the commit point BEFORE the specified commit-hash
* Useful in collaboration. When multiple co-workers/collaborator has the same bad commit, it will cause a lot of trouble if you just reset the bad commit. Since many sets of work are based on the bad commit, so it will be difficult to reconcile those changes if the commit is entirely removed using <git reset>
* Git reset can be used if you’re just working on a local repository.

**GitHub**

**Markdown**

Useful link: markdown-it.github.io

* Contains all syntax of markdown and shows effects locally

**GitHub: The Basics**

What does GitHub do for us?

* Hosting platform for git repositories, in the cloud (used for sharing, collaborating etc.)
* Serves as a backup for your git repositories as well

Why You Should Use GitHub!

* Open Source Projects -> code base is available for viewing, contributing and fixing bugs
  + Make contributions if you are planning on finding a job in the area eventually
* Exposure
  + Can act as a resume that employers will consult in the hiring process
* Stay Up To Date
  + Stay active in the community, be ready and anticipate changes in certain programmes etc.

Cloning

* Important part of working with remote repositories (repos hosted on the web.
* Downloading the content of a remote repository onto your local machine
  + <git clone (url)>
  + Make sure you’re not in a repo when you clone!