**Git Notes**

**Key Terms:**

* Commit: Snapshots of files that enables the contributor(s) to go back to a certain snapshot of a file, effectively cancelling out the changes made AFTER that snapshot was taken. Basically, a commit saves the state of the files at a particular point in time.
* Working tree: What we see on our filesystem (FS). Adding, editing and deletion of files happens in the working tree. Usually the working area is the directory that we converted into a Git rpeo using the “git init” command (without the quotes).
* Staging Area (Index): The area that stores our files BEFORE we make a commit. This area gives us full control on what we want to commit. Only the files we put into the staging area are put into the next commit.
* Git History: The commit history. It is located in a hidden directory called dot git (.git)
* .git directory: Holds an object database and metadata that makes up our repo.
* Untracked file: A file that’s manually added to the Git repo. Since we manually added the file, Git doesn’t know that we added a file and since Git tracks files over time, you need to manually make Git “track” the new files you’ve added.
* .gitignore: Sometimes we don’t want Git to track some files. For example, log files, compiled code and non critical stuff. We can make Git not track these changes by placing these files in the .gitignore text file. Create and open the .gitignore text file and put in the FULL name of the files (if directories put ‘em in the following format: DIRECTORYNAME/) that we don’t want Git to track. Don’t forget to commit this!
* Branch: Allows us to work on different versions of the same files in parallel. Edits on one branch can be independent of work on other branches. We can later merge our changes into other branches (most of the time it is the “master” branch.)
* HEAD: A pointer that normally points to a branch.
* Merge conflicts: A merge conflict occurs when we try to merge branches that have changed the same lines in the same files. During conflicts, if one branch has changed a line in a file and the other branch didn’t change that line in that same file, Git will change the line as in Git terminology, in a change vs no-change, the change wins. After solving the conflict, you can proceed as normal: Stage it with **git add .** and since it does a 3-way merge, you have to write a merge commit with **git commit**.
* Detached HEAD state: This situation occurs when the HEAD pointer is pointing directly to a commit instead of a branch. To get out of this situation, you can simply write either of these commands: **git checkout master** (or any other branch you might have) OR **git branch BRANCHNAME** AND **git checkout BRANCHNAME**.
* If we don’t have a clean working tree, we can either commit those changes or use a command called **git stash** which saves our changes to the file we edited so we can apply them later. If we check with **git status** it should show a clean message with “working tree clean.”
* Git Remote: A repo in another location from where we are currently working, usually GitHub. From the perspective of our laptop, GitHub is a remote repo. If there are changes to the repo at a remote, we can download those changes to get them to our laptop. If we make changes to the “local” repo (the one we created with Git on our laptop), we can upload those changes to the remote repo. The remote could be a copy of “fork” of a repo within GitHub, a repo on a co-worker’s system, or even an another directory on our same “local” system.
* While “fetching” from or “pushing” to GitHub, we don’t need to specify the location (https://github.com/YOUR-USERNAME/YOUR-REPOSITORY). Git can hold an alias to that location. The default alias is called “origin”. The “origin” will point to our remote repo at GitHub.
* When we run the command:

**git log --all --decorate --oneline --graph**

while working with remote repos, you can notice that we have something new: “origin/master” and “origin/HEAD”. origin/master is a specialized branch. It’s called a remote-tracking branch. It tells us what the master branch looks like at “origin”.

* Fork: A copy of a project at GitHub by someone else. You can “fork” their work and work on it yourself and “push” changes to your master. After you have created something and want to merge your work with the real project, you can create a “Pull Request” which the original author may accept and as a result merge your changes or work into their master.

**Git Commands and Purposes**

**Note: Do NOT put quotes in your command line as the command line would throw an error. Exceptions to this are the**

**git commit -m “WHAT DID YOU DO/DESCRIPTION OF THE COMMIT” command, the git config --global user.name “YOUR-NAME” command, the git config--global user.email “YOUR-EMAIL” command and the git stash save “MESSAGE” command.**

**Core/Basic Git Commands**

|  |  |
| --- | --- |
| Git Commands | Purpose |
| git init | Turns a directory into a Git repository (repo). To check if it worked see if a new hidden subdirectory called .git is added. |
| IMPORTANT: git config --global user.name “YOUR-NAME” | Sets your name in Git so that when commits are made, it displays who made the commit and when. |
| IMPORTANT: git config--global user.email “YOUR-EMAIL” | Sets your email in Git so that when commits are made, it displays the email of the person who made the commit and when if any contact needs to be made with that person regarding their commit. |
| git config --list | Shows the username and email set in Git along with some extra info. |
| git status | Tells us how things stand in our working tree and in our staging area. Useful to know whether we have some untracked file. |
| git add FILENAME | Adds our untracked and modified file specified with FILENAME into the staging area and makes it a tracked file. For example: git add file1. **Note: You MUST do this after every file you add or modify inside of the Git repo (the directory we changed into a Git repo by using git init)** |
| git commit -m “WHAT DID YOU DO/DESCRIPTION OF THE COMMIT” | This is the shorter and easier way to commit. Commits our changes in the Staging Area to the Git repository. **Note: You MUST do this after every file you add or modify inside of the Git repo (the directory we changed into a Git repo by using git init) after you used the git add FILENAME command.** |
| git log | Shows us our commit graph. |
| git diff | See the differences between tracked files in the working tree and the staging area. Doesn’t work for untracked files. Returns nothing if the working tree and the staging area match. |
| git add . | Automatically adds ALL our untracked and modified files to the staging area. **Note: You MUST do this after every file you add or modify inside of the Git repo (the directory we changed into a Git repo by using git init)** |
| git diff --staged | Shows the difference between the Staging Area and the most recent commit. |
| git log -p | Shows us commit graph along with what’s different about each commit. |
| git rm FILENAME | Removes the file specified with FILENAME and also stages the removal only needing to commit the change. For example: git rm file1 |
| git commit | This is the longer way of commiting. Opens your default text editor and makes you to describe the commit. An advantage of this is you can write a detailed, multiline commit message. |
| git checkout -- FILENAME | Replace the new, often bad or corrupted FILENAME in the working area, with the previous version of the FILENAME in the staging area, effectively discarding the new working tree changes. |
| git reset HEAD FILENAME | Restores the version of FILENAME from the most recent commit into the staging area, effectively unstaging the new, often bad or corrupted FILENAME. **Note: This just restored the changes in the staging area and NOT in the working tree. Use the**  **git checkout --FILENAME command to restore the change in the staging area into the working tree.** |
| git log -- FILENAME | Shows the commits that affect FILENAME. |
| git checkout COMMIT-HASH -- FILENAME | **Note: Replace “COMMIT-HASH” with the first-five hash characters (hash of the specific file can be retrieved by the command:**  **git log --FILENAME).** Recovers a deleted FILENAME from a prior commit that has FILENAME. It also puts FILENAME in the working tree and also in the staging area it to be committed. |

**Branching, Merging and Stashing**

|  |  |
| --- | --- |
| Git Commands | Purposes |
| git log --all --decorate --oneline --graph  Note: Since this command is very long, but very useful, you can create an alias to shorten this command in Git Bash by writing:  alias graph='git log --all --decorate --oneline --graph'  Note: I chose the alias name as “graph” but you can put whatever is easiest for you. | Shows us a nice and labelled commit graph. Also shows us to which branch the HEAD pointer is pointing towards. |
| git branch BRANCHNAME | Makes a new specified branch (BRANCHNAME). |
| git branch | Shows all of our branches. |
| git checkout BRANCHNAME | Switches to the branch specified branch (BRANCHNAME). Also moves the HEAD pointer to the specified branch (BRANCHNAME). |
| git commit -a -m “WHAT DID YOU DO/DESCRIPTION OF THE COMMIT” | Stages and commits any tracked files that have been modified. |
| git merge BRANCHNAME | Merges BRANCHNAME to the checked-out branch. It’s a good idea to move to the master branch before merging. **Note: when doing a 3-way merge, since it is a merge commit, we need to enter a commit message.** |
| git diff master..BRANCHNAME | Shows what will change when merging BRANCHNAME into master. |
| git branch --merged | Shows which branches are merged with the branch we are currently on. |
| git branch -a | View both “local” and remote branches. |
| git branch -r | View only remote tracking branches. |
| git branch -d BRANCHNAME | Deletes the specific branch (BRANCHNAME) |
| git branch -D BRANCHNAME | Deletes the specific branch (BRANCHNAME) if it isn’t merged to the branch we are currently on. |
| git checkout -b BRANCHNAME | Creates and checks out (switches) to the created branch (BRANCHNAME) |
| git merge --abort | Aborts a merge between two branches during a conflict. |
| git stash | Saves our changes to the file we edited so we can apply/commit them later. |
| git stash list | See the stashes/changes we saved. |
| git stash -p | Observe the edits that occurred with each stash point. |
| git stash apply | Applies the most recent stash. **Note: The applied stash isn’t staged and hence not committed yet. The stashes aren’t removed after we have staged and committed the changes.** |
| git stash pop | Applies the most recent stash and removes that stash. |
| git stash apply stash@{STASH#} | Applies the specific stash with the stash@{STASH#} rather than the most recent one. **Note: Replace “STASH#” with the stash number you want to apply. To find out which STASH# you want to apply, run the** git stash list **command and you will find the STASH# on the left.** |
| git stash save “MESSAGE” | Leaves a message to make the stashes more helpful and easier for us to know what that specific stash contains. **Note: In the “MESSAGE” section you need to write a message. If you write the command like this, “git stash save “MESSAGE”, when you run the “git stash list” command it will just show MESSAGE as the message** **you left for the stash which isn’t helpful at all.** |

**GitHub (Remotes)**

|  |  |
| --- | --- |
| Git Commands | Purpose |
| git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY | Clones the specified project to the directory you are currently on and creates a new subdirectory with the repo’s name. |
| git remote | Display’s our remotes. |
| git remote -v | Shows us the full location(s) of the remote(s). |
| git fetch origin | To be updated with the changes at the “origin” or GitHub. **Note: If you want to see the changes in detail, run the git status and/or the alias git log --all --decorate --oneline --graph**  **commands.** |
| git merge origin/master | Merge the changes done at the “origin” or GitHub into our “local” repo; in other words, merge the changes in GitHub to whichever directory we cloned the “origin” to. |
| git pull | Combines git fetch origin and git mergin origin/master into a single command. (I would advice to stick with the separate fetch and merge commands.) |
| git push origin master | Uploads or “pushes” our changes from our “local” repo to the remote named, “origin” or GitHub, and to the master branch at “origin” |
| git remote add upstream https://github.com/ORIGINAL\_  OWNER/ORIGINAL\_REPOSITORY.GIT | For Forking: Add a new remote that will keep your fork in sync when called upon. |
| git remote remove REMOTENAME | Deletes the specified remote (REMOTENAME). For example:  git remote remove upstream |
| git fetch upstream | For Forking: To be updated with the changes at the “upstream/master”; In other words, run this command to be in sync with the master branch of the repo you have forked. |
| git merge upstream/master | For forking: Merges the most recent commit from the upstream/master branch, into our local master branch. |
| git push origin BRANCHNAME | For forking: Uploads or “pushes” the new branch (BRANCHNAME) to the “origin” or to the fork we have at GitHub. |