t

Git Commands

* Some Linux command used in it git for directory manipulation
  + **pwm:** present working directory; shows in which directory we are in.
  + **ls:** list the content of the directory we are in.
  + **cd:** change the directory
  + **touch <file name>:** to create a file.
* **git status:** shows the status of the directory we are in.
* **‘git add .’ / ‘git add - - a’:** to add files in the staging area.
* **git commit -m “message”:** commit the file with message to git
* **git commit:** This will take us to the editor in which we will write the commit messge and then commit the file.
* **git commit –a –m “message”:** It will stage all the tracked modified files and commit them. But untracked files will not get commit we have to first track them.
* **git commit --amend:** The command is used to make changes to the most recent commit in your Git repository. It allows you to combine staged changes with the previous commit, effectively modifying the commit message or the content of the commit itself.
* When we enter this command git take us in the editor where we have to first press **‘I’** to insert message and then press **‘Esc’** and then press **‘shift+:’** to enter incommand mode in editor and them press **‘wq’** to save the changes and exit the editor

**Restore and Reset git -:**

* **git checkout -- <file name >:** This command is used to revert back the changes made in a tracked file to the most recent commit. Already staged file can’t be revert back to most recent commit ; it also act like a safety feature for scenarios like in which we accidently revert back a file.
* **git checkout -f:** This is used to revert back the whole working directory back to most recent commit of the working directory.
* **git restore --stage <file name>/** **git reset HEAD <file name>:** To unstage the already staged file.
* **git reset HEAD~n:** It is used to revert back the entire project from the latest commit to the older commits. Like for reverting project by one commit we will write 1 inplace of 'n' and by 2 commits we replace 'n' by 2 and so on.
* **git reset --hard HEAD~n:** It works same as this command **git reset HEAD~n** but it will also **delete any changes** added to the project since the n commit.
  + We can also Reset to a specific commit using its hash too. Like this **git reset --hard <commit hash>.**
  + We can also reset the entire branch to the state of the specified branch. Like **git reset --hard <branchname>**.
* **git diff:** It will compare the current working directory to the current staging area.
* **git diff --staged:** It will compare the previous commit to the current staging area.

**Remove and rename -:**

* **Rm -rf .git:** to remove the directory from git.
* **git rm <filename>:** Deletes the file from the working directory. But if we want to delete a file in staging area we have to write command as **git rm <filename> -f** .
* **git rm –cached <file name>:** The file get untracked . But it will still show as it get deleted in git, but actually it still in directory just isn’t getting tacked by git.
* **git mv <old file name> <new file name>:** It is used to rename a file, but what it actually does is that it moves old file content to new file so it is used for both rename and move one file content to another. It will directly add the file to staging area.
* **Q :** by clicking Q we can exit from different options.

**git log and its filter -:**

* **git log:** Shows all the commits done in the working directory.
* **git log –p:** Willshow all the difference/changes too with the commits. To see only specific commits like last 2 commits etc. what we’ll do is we write command as **git log –p -2** and git show only last 2 commits done in the directory. We can see as many commit we want we just have to write that specific no.in the **git log –p –n** command.
* **git log --stat:** It will show all the changes and commits in short summary..
* **git log --since=<time>:**It will show all the commits done in specific time period like 2 days, 2 months, 2 weeks etc. to see the commits we have to write 2.days in place of time in command for 2 days and git will show all the commits happen in this time period
* **git log --pretty=<format>:** Shows the commit log in the format we want to, by just writing that format in command like these formats also called filters:
  + **git log --pretty=oneline:** It will show each commit log in single line which makes it easy to copy and so on.
  + **git log --pretty=short:** It will show commit log with some exempt information.Like it will only show author name..It can make searching a specific commit easy.
  + **git log --pretty=full:** It will show more info like author name, message ,and committer name.

**Author is who created the file first and committer is who make that change in the file.**

* + **git log –pretty-format:”%h -- %an”:** It will show the git log in specific way like this db66c2a – Saurabh where %h is short hash commit and %an is author name and these are known as placeholder. We can write any placeholders in the string like %ae: author email etc. .{ *There are many formats we can write but we can remember as they are obscurely used tedious to remember so we refer this git site* [**https://git-scm.com/docs/git-log**](https://git-scm.com/docs/git-log)*to see the commands as we needed*.}
  + All these commands are used very rarely mostly in complex workflows normally we work with these commands mostly **git log, git log –p –n.** If we have to work with these we can also refer the git site too.

**Remote repository -:**

* **git clone <url name> <file Name>:**clones a remote git repository.
* **git remote:** Shows the added remote repository in working tree.
* **git remote add <repository name> <url of repository>:** It adds the remote repository url to the working tree. We can give any name to the remote repository but we normally give origin name to it like; <https://github.com/Saurab-h/git-prcatice.git> .
* **git remote -v:** It will show the both pull and push remote repositories like below

$ git remote -v

origin https://github.com/Saurab-h/git-prcatice.git (fetch)

origin https://github.com/Saurab-h/git-prcatice.git (push)

In this case both pull and push remote repository are same but they can be different too.

* **Git push -u < repository name > <branch name>:** It pushes all the changes and tracling files of working tree to the remote repository.
* **'-u'** sets the upstream tracking branch, which means that in the future, you can simply use '**git push'** without specifying the remote and branch names.
* To push the changes to the repository we had to have the permission (To protect the repository from unwanted manipulation by others).
* To get the permission we have go to the settings of the site we have made the remote repository .
* Then go in the SSH and GPG keys option and clicl new SSH key
* Fill the title and key box.

In the key box we have to fill the SSH key associated with our git account.

* To generate the SSH key go to this site [**https://rb.gy/u6wtz**](https://rb.gy/u6wtz) **(**a shortlink of github generate ssh key**)** and follow all the steps mention in it and your key will get generated.
* **git config --global alias.<new command> <'old command'>:** It help us giving a command an alias name so that instead command we can use alias command for the easiness of writing commands. Like:

**git config --global alias.unstage 'restore --staged --' =>** Thorugh this we have changed the **restore --staged --**  to unstage so that we can unstage a file by git unstage <file name> and don't have write the whole command . But still we can use old too , so both command is valid for unstaing the file. (The -- is used in Git to separate the end of the options from the beginning of the filenames. In this case, it is necessary to ensure that if a filename starts with a hyphen (-), it won't be mistaken for an option. Including -- after --staged is a best practice to handle filenames with leading hyphens properly).

**Branches -:**

* **git checkout -b <new branch name>:** It will create the new brach and take us to the same branch.
  + Here **'-b'** instructs Git to create a new branch with the specified name.
* **git checkout <branch name>:** Takes us to the branch we mentioned in the command e.g. **git checkout master** will take us to master branch and so on.
  + The files in .**gitignore** will get ignored by git will not have any effect of changing branches and will stay in every one until we take them out from **.gitignore**.
* **git branch:** Will show every branch in working tree and show the current branch we are in like this **\* sample branch .**
* **git merge <branch name>:** It will merge the mentioned branch into the branch we are working in.
  + When we change the same the thing in both the branches we want to merge the git will show us the **Merge Conflict** message and ask us to choose which we want to keep by editing back the change we don’t want from one of the branches(VS code gives us a feature to easily resolve ths conflict directly in the VS code editor). After resolving the conflict we will go to bash and stage & commit the change in the branch and merge will be completed.
  + This merge conflict because git thinks both changes are equally important and can't choose which change to choose so it asks us to choose the change we want to keep.
* **git branch -v:** It shows all the branaches with there respective branch hash and last commit of each branch in the working tree.
* **git branch --merged:** It will show all the already merged branches.
* **git branch --no-merged:** It will show all the branches which are not yet merged.
* **git branch -d <branch name>:** With it we can delete the branch which is mentioned in the command.
  + But when we try to delete a branch which is not yet been merged and have changes in it the git will show us an error to prevent the accidental loss of data, but if we still want to delete the branch we can do se by typing this command **git branch -D <branch name>** and git will delete the branch.
* **Terminal shortcuts:**
* **Ctrl insert:** Copies the selected content.
* **Shift insert:** pastes the selected content.
* **Tab:** Auto completes the file name if same name doesn’t exist anywhere in directory.

Ignore file**:** To ignore files/directories in git we just have to create a “**.gitignore**” file and write the name of those file and directories in it to ignore.

* To ignore multiple files of same type we should write \*.extension to ignore all of them for e.g. To ignore all .txt files we just write “\*.txt in **“.gitignore”** file.
* We can ignore the whole path, single directory or multiple directory of same name or a specific file from the multiple files of same name.
* To ignore all the files with the same name have write file name like this (dir/) in “**.gitignore**”, so to ignore a file of a same name as other we have to specify the path of the file to ignore it.(for outer: /dir/, for inner e.g. static/dir)
* Git already ignores the blank folder automatically and if we put an ignored file/folder in it, it will still ignore it. If we add a file in blank folder git will recognize it but still ignore the ignored file.
* The **.gitignore** file allows more complex patterns, such as using wildcards, negation, and comments. Feel free to explore these possibilities for more fine-grained control over what to ignore.

Remember, the **.gitignore** file is an essential tool to keep your repository clean and avoid tracking unnecessary files. While the above tips cover most common use cases, you can tailor it to suit your specific project needs.

By following these guidelines, you can efficiently manage your Git repository and focus on tracking only the essential files while leaving out the temporary and generated files, build artifacts, and other non-essential elements.

Note: If we ignore already tracking file it will still be get tracked by git, to ignore the file first we have explicitly make it untracked with the help of **git rm –cached <file name>** command.

Git File status



**File status life cycle:** When we track the untracked files first time instead of saying that they are in staged we say they got unmodified (just for the first time), after that whenever we modify these files they will become modified, after which whenever we track these file again they will go to staged and then when we commit the files they will become unmodified again.

* And when we modify a file which is still in the staging area (the area in which ready to commit files are placed.) then it will be present in both staging area and modified area, but in staged the version of file is before it get modified which is ready to get commit and in modified area new modified file is present.

When we commit the file, the file in staging will get committed not the modified one and we will still have the both version of the file (that’s why staging area is used to avoid these kind of situations).

But if we add the modified file in staging area it will merge with the already existing one in the staging area and will become the only version of file present in git.