**GIT**

* It is a distributed version control system which uses widely among other version control systems.
* A version control is a software tool which manage the changes to source code over time.
* It keeps track of every change made and provides proper description about what changes have been made in the specific version and when it was changed.
* If a developer wants to use earlier versions of code, you can use it anytime with the help of version control.
* When you add new files and commit them, it will store in local repo. Only after you push, it will go to remote repo.
* **git init** = to initialize repo in local machine.
* **git config –global user.name “name”** = Set a name for your commit transactions.
* **git config --global user.email “email”** = Set a mail for your commit transactions**.**
* **git clone repo-url** = To clone remote repo in local.
* **Remote/origin =** it is some git repository on github.
* **git add file** = To add file to staging area (or) index.
* **git status** = To see which files are in staging area.
* **Git rm filename –cached = to unstage a file from staging area**
* **Git rm filename –f = to remove file completely after adding**
* push to origin to delete the file in remote repo.
* **git add \* =** To add all files which are in your branch at once.
* **git commit –m “commit msg”** = To commit changes to repo.
* **git commit --amend –m “commit msg” =** To change the commit msg of last commit.
* **git show commit-id ==** to see contents of specific commit**.**
* **git push –u origin master** = To push changes to remote repo master branch.

To push to other branch…type branch name in the place of master.

* **git pull origin master** = To pull changes from master remote repo.

To pull from other branch…type branch name in the place of master.

* **Git fetch** = It is used to download data from remote repo, but it doesn’t integrate(**merge**) any of this new data into working files.
* **Git pull** = it is also used to download the data from remote repo, but it integrates(**merges**) the new data into working files.
* When you **clone** git remote repo, it will clone only master branch to local, to get all branches to local type **git branch –a** to show all branches.
* If you want to work on local branch, you need to create local tracking branch.

**git checkout –b branchname origin/branchname**.

* Or You can just checkout to the branch with **git checkout branch** with out creating a local tracking branch.

**DIFF:**

* It is used to compare multiple version of files. It can compare between commits, branches.
* **git diff HEAD file** = comparing file from HEAD with unstaged file.
* **git diff** = same use as above command.
* **git diff --staged** = compares all staged files with HEAD files.
* **git diff --staged <file>** = to compare only one staged file with HEAD.
* **git diff --cached =** shows same output as --staged command.
* **git diff commit-id1 commit-id2 =** shows differences between two commits**.**
* If you have commited multiple files at once, it will shows differences in all files.
* **git diff commit-id1:file commit-id2:file =** shows differences between a file from two commits
* **git diff branch1..branch2** = shows same output as above command.
* **git diff branch1..branch2 file** = shows difference in a specific file from two branches.
* **git diff branch1:file branch2:file =** shows same output as above command.

**BRANCH:**

* **git branch** = To see local branches.
* **git branch –r =** shows remote branches.
* **git branch –v =** shows branches with latest commit ID and commit message**.**
* **git branch branch-name** = To create a local branch.
* After creating a branch in local, push from that branch to be applied in remote repository.
* **git checkout branch-name** = To go to that specified branch.
* **git checkout –b branch =** Creates a new branch and checkouts immediately**.**
* **git branch –d branch-name** = To delete a local branch.
* **git branch –D branch =** To delete a local branch forcefully**.**
* **git push origin --delete branch =** To delete a remote branch**.**
* **git merge branch-name =** To merge two branches.
* You should be in the destination branch in where you want to merge the src branch. The files in the src branch will be merged to destination branch.
* When you merge branches, git will create a new commit from the merged.
* **git branch –m old new =** To change branch name.
* **git branch --merged** = To see which branches have merged into current branch.
* **git branch --no-merged** = To see branches which have not merged into current branch.
* **Rebase =** rebase will merge and apply the changes at top of the branch instead of creating a new merge commit.
* **git rebase branch =** apply the commits at the top of the history without creating a merge commit.
* **git rebase --abort** = To abort (or) stop the rebasing when an conflict occurs.
* **git rebase --continue =** To continue the rebase from the last, after you fixing any conflicts.
* **git log** = To see version history with commit id, commit msg, date and time.

**STASH:**

* stash takes your current working dir and puts in a stack for later use and gives you back a clean working dir.

**EX** : if you are working in a branch, half of the work is completed and you want to go to another branch and you don’t want to commit this work. So we move this files to stash. It will store files in a stack. If you didn’t stashed (or) commited the files and switched to another branch, the files in the old branch will be deleted.

* Stash works only on files, which are in staging are (after adding).
* **git** **stash** = to save files in a stack. It will store with your previous commit id.
* **git** **stash** **list** = To see stashed files.
* **git stash apply** = get back the files from stash to branch. But files will present in stack too.
* **git stash drop =** To drop files from stack (delete from stash)**.**
* **git stash pop ==** apply and deletes files from stash**.**
* **git stash apply stash@{0}** = To remove a specific file from stashing.
* **Stash list : .git/refs/stash latest stash id will be stored**

**SQUASH:**

* Merging multiple commits as one single commit with the help of git rebase. You can do squashing before pushing changes to remote repo.
* **git rebase –i HEAD~3 =** To merge last 3 commits**.**
* It will show you the configuration file to choose which commit should retain and which commit should be squashed.
* For ex:

**pick commitID commit-msg**

**Pick commitID commit-msg**

**Pick commitID commit-msg.**

* By default, It will show you the commit order from old to new.
* In the above example, I have 3 commits and I want two commits to be squashed.
* To do that, remove **pick** and type **s** before the 2 commits to which you want to squash and leave **pick** before the commit to which you want to retain and Save the file.
* After saving, it asks you to edit the commit history by opening a file and shows you the commit messages.
* Remove the 2 commit messages from the file that you want to squash and leave the one commit, which you want to retain.
* After squashing, if you push changes to remote branch you will get an error. Because, your tip of the branch is now behind. So, push the changes forcefully with **“–f”**.

**RESET:**

* The command to revert the changes to files is called **RESET**.
* There are 3 types of reset commands.
* **Soft** == it just moves the HEAD of the branch to the specific commit. The data goes to staging area.

**git reset --soft**

* **Mixed ==** it moves the HEAD to previous commit and resets the staging area. This is the default reset type git uses when you use reset command**.**

**git reset --mixed**

* **Hard ==** it moves HEAD to previous commit and deletes the files which are commited after. It will remove the commit from history.

**git reset --hard**

**REVERT:**

* **Revert** command works just like **reset** but with only one advantage. Instead of deleting commit history like reset, revert will save as new commit with the changes.

**git revert commit-id** == it will undo the changes to that specific commit.

**git revert --abort** = to abort the reverting.

**git revert --continue** = to continue reverting after fixing conflicts.

* With the help of **checkout** command, you can travel between commits and repository will show you the content upto that commit.

**git checkout commit-id** == to go back to specific commit.

* To go back to your branch just checkout with your branch name.
* If you want to take this commit to a new branch, create a new branch from this commit by typing **git checkout –b new\_branch**. It will create a new branch with the HEAD pointing to this commit.

**TAGS:**

* Git has an option to tag a commit in repository history so that we can find it easier in the later point of time.
* This tag acts as a reference to your commit to identify easily.
* To apply a tag to commit = **git tag –a <pattern> -m “message” commit-id**
* To see list of tags = **git tag**
* To see contents of a tag = **git show <tag\_pattern>**
* To remove tag = **git tag –d <tag\_pattern>**