To configure username to git

-> git config --global user.name “user\_name”

To config emailto git

-> git config --global user.email “Email\_id”

To see details in git which are saved

-> git config --list

To get username

-> git config user.name

To get email

-> git config user.email

**Git - 3 Stage Architecture**

-> if an application is developed by us and names as version1

-> we want to make changes in the application but because of the fear that if anything goes wrong then we must be able to restore v1, it should not be that we lost our previous working model

-> so we want that a snapshot of the v1 is taken and saved and whenever we need it it can be used

-> this process of taking snapshot as **commit.**

-> if we make changes in 3 of the folders and out of those change in 1 folder leads to some errors

-> now we want that we can pick the data of the faulty folder from the previous version and update the rest folders

-> **working directory :** the directory of project

**-> staging area:** files to be included in next commit

**-> git directory(repository):** folder with .git extension, contains compressed files

-> git status : tell about whether the current folder is a git repo

-> git init : it makes the current folder an empty repo

-> **git add --a** and **git add .** : it sends the files of the current folder into the staging area of the current git repo

-> git add file\_name : to add a particular file to the staging area

-> git commit -m “message” : it commits the files of the current folder into the repo. Here -m denotes that I want to commit with just a message and without opening the editor

-> git commit : it will open an editor

-> git log : helps to know the commits that were carried out

**Note:**  if we run git init twice it will reinitialize the repo which is not wanted

-> rm -rf .git : it removes the files from the current folder and deletes the repository. It removes the .git folder. Therefore no tracking is done

-> git clone link : it clones the repository on ur system

-> git clone link custom\_name : it clones the repo into ur system with name specified in custom\_name

**-> .git file :** it contains version control info. It contains all the versions of repo and will return the asked version when git command is run.

**Note:** When “git init” is executed it makes the folder enter into the “untracked” area.

**Note:** When a folder is untracked and after “git add --a” command is executed it does not enter into the staging area but its tracking starts. However after this whenever “git add --a” is executed it sends the files to the staging area. Thus “git add --a” is a multi-purpose command.

**Note:** If we have an error.log file in our software that saves all the error that are experienced when using the software then we don’t want our repo to be committed when an error comes. So we want that our repo should ignore the error.log file

For this we create a file called “.gitignore” using command “touch .gitignore”

We put all the files and the folders in “.gitignore” that we want to ignore and don’t want commit

**Note:** if we have added a file in staging area and then we modify it then there will be 2 version of the file- 1 in the staging area, 2-in the modified area

-> git diff: it compares the staging area and the working directory

**Note:** if the modified file is added to the staging area then the version of the file in the staging area will be merged with the version of the modified file and now only 1 version of the file will exist ie the modified one.

Thus “git diff” will show nothing

-> git diff --staged : it helps to compare the last commit with the current staging area.

**Note:** in commit message we should add info like what changes are made in the project and whats the purpose of making those changes

-> git commit -a -m “custom\_message” : helps to skip adding the **tracked** files into the staging area and directly committing them. It **does not** work for the **untracked** files. The untracked files need to be added to the staging area first to get started to be tracked. Thereafter we can commit it skipping its addition to the staging area

**Note:** if we rename a file in our folder of repo and execute the “git status” command then it will show that the file with previous name is deleted and file with newer name is added but is untracked. When we add the untracked file using “git add .” command then it comes to know that we didn’t delete the file but just renamed it. Then we can commit this change. Also if we delete a file from the folder and execute “git status” command, then it will show that the file is deleted, but if we restore the file into the folder back again, then on executing the “git status” command it will show no changes are there or to be made.

-> git rm “file\_name” : helps to remove the files directly instead of following the process described above.

-> git mv “older\_file\_name” “new\_file\_name” : helps to rename the file from “older\_file\_name” to “new\_file\_name” and will also stage this automatically.

**Note:** if we are tracking a file and then we add it “.gitignore”, and then we modify it, “git status” will show that the file is modified even though changes in “.gitignore” file are committed to ignore that file. This happens because the file was being tracked since earlier and thus any changes in the file even after adding to “.gitignore” are shown by bash. To avoid this problem we need to untrack the file

-> git rm --cached “file\_name” : it will untrack the file being tracked, and will not delete it. After this if we edit the file and check the status it wont show anything as its been added to “.gitignore” file.

-> git log -p : it display the commits and also the changes made in the commits.

-> git log -p -”n” : displays the last “n” commits with the diff(changes).

->git log --stat : it displays the changes made in the commits in concise manner

-> git log --pretty=oneline: used to print all the commits on a single paper. It gives the info about all commits in one line

-> git log --pretty=short : gives info about all commits in very short. Does not tell about the author and the person who made the commit

-> git log --pretty=long : tells about the author and the person who made the commit.

**Note:** Author is the one who created the file and the Committer is the one who made changes into the file and commit those changes .

-> git log --since=2.days : tells about the commits made in the last 2 days

-> git log --since=2.months : tells about the commits made in the last 2 months

-> git log --since=2.years : tells about the commits made in the last 2 years

->git log --pretty=format:”%h -- %an”: logs according to hash commits and the author name

Rest filters can be seen from the git documentation.

-> git commit --amend : to merge changes in the last commit

-> git restore --staged “file\_name” : is used to unstage a file from staging area

-> git checkout -- “file\_name” : used to undo the changes made in the file and restore it to the version of the last commit of the same file

**Note :** once the modified file is added to the staging area and after that we execute the “git checkout” command then it wont work and wont undo the changes made in the file.

-> git checkout -f : it is used to undo the changes made in all the files that were modified and is restored to the version of the last commit

-> git remote: it is used to check all the remote repo’s we have.

-> git remote add “name of the website containing the repo(usually origin)” https:”url of the repo”: it is used to add a remote repository like a repo made in GitHub

**Note:** GitHub is a website that hosts the git repo’s

-> git remote -v : it tells about the remote repo we have and the url for fetching data from it and url for pushing changes to it.

-> git remote rm “name of the remote repo” : it is used to delete the specified remote repo

-> git push : it is used to upload local repo content to a remote repo. Its counterpart is fetch which imports commits of remote repo to local repo

-> ls -al ~/.ssh: it is used to check whether there are any existing ssh keys.

**Generating New SSH - Key Pair (Public - Private):**

-> $ ssh-keygen -t rsa -b 4096 -C ["your\_email@example.com"](mailto:\"your_email@example.com\") : is used to generate new ssh key

**Starting the ssh-agent:**

-> $ eval $(ssh-agent -s) : it starts the ssh agent in the background

-> $ ssh-add ~/.ssh/id\_rsa : it adds the private ssh key to the ssh-agent.

-> $ tail ~/.ssh/id\_rsa.pub : it is used to get the private ssh key

-> git pull origin YOUR\_BRANCH\_NAME : it is used to overcome “non-fast-forward” errors

-> git push -u origin master: used to push the local repo to the remote repo under master branch

**Note:** alias are used to make longer git commands given shorter names so that they can be used easily. Eg: if we want to run “git status” command as “git st”, then alias will help us in getting this job done

-> git config --global alias.st status: it will make st equivalent to status. Thus if we exec “git st” it will work similar to “git status”.

**Note :** Branches are very important when websites are in production and wants to make changes in the code of the website. Initially we have a master branch and the code of master branch is in production and is thus deployed. Now we want to make changes in the code but if we make changes in the master branch then might be our website goes down because of the errors that will occur while making changes in the website. So to prevent this we create a branch out of master which will be containing all the code present in the master. Then we can make changes in the branch without disturbing the original code and after changes are made and all things are working correctly we can merge the changes of this branch with the master branch and deploy the modified version of the website.

-> git checkout -b develop: it is used to create new branch(-b) named “develop” and takes us into that branch

-> git checkout “name of the branch”: used to switch bw branches

-> git branch: it will show all the branches present. The current branch will be highlighted.

-> git merge “branch name”: to merge the branch with the master branch

**Note:** we need to commit after we merge the branch with the master branch as it will save the changes made to the master

**Note:** if we create a branch out of the master branch and start making changes in the application and simultaneously changes are being made in the master branch then at the end if we try to merge the changes made in the branch into the master before committing the changes made in the master , then a merge conflict will occur saying that its not understandable which changes to be kept either the branch changes or the master changes. Here it asks the user to tell which changes are to be kept.

**Note:** git stores the changes made in the new version and uses a pointer that points to the last commit

-> git branch -v: shows info about the branch and its last commit hash and commit message

-> git branch --merged: will display the branches that are already merged

-> git branch --no-merged : will display the branches that are not merged

-> git branch -d “name of the branch”: it will help to delete the branch but will give an error saying that branch is not merged and thus will warn that u r trying to delete the branch that is not merged yet.

-> git branch -D “name of the branch”: will delete the branch without any error and warning

**Note:** branches are not pushed automatically when “git push” is executed. We need to do this explicitly.

-> git push “remote name” “branch name”: is used to push the branch we created and did not get pushed automatically.

**Note:** when we are trying to push a branch then it is recommended to be in the same branch we are trying to push.

-> git push “remote name” “branch name”:”branch name with which we want to track the branch in remote” = is used to specify the name of the branch with which we want to track our local branch in remote.

-> git push -d “remote name” “branch name”: to delete the branch from remote repo.