**Version Control Tools**

They are the software which track and manage the changes happening inside our files.

* Make save points, so that we can go back to any save point at any point of time.
* Because of this they provide total development freedom. For ex- we have developed version 1.0 ,all the functionalities are working in that version, so we make savepoint there with the help of git after that we developed 1.1 version but we saw all functionalities are not working there so now because of git we can go back to last savepoint where all the functionalities are working, because of this feature we have total development freedom.
* E.g- Git, subversion, perforce.

**Note** -When the programmer commit(save) the changes it becomes savepoint.

**GIT**

It is a version control tool.

**GITHUB**

It is a online-service, where we can host our repositories. (just like Youtube is a video hosting service), and we are doing this so that multiple people can see,access and contribute to our repository.(collaboration becomes easy).

**GIT REPOSITORY**

It is a folder where changes are by the git.

**DIRECTORY**

Means folder.

**GIT LIFECYCLE**

**Stages of a file in Git**

1. Working Directory(Untracked files)
2. Staging Area(Tracked files)
3. Git Directory(Permanent Save)
4. Working Directory

Working directory is the folder where files are stored and these files are monitored by git means git is aware of the files but tracking has not been started by git for these files yet.

So if we want to convert our directory into working directory we have to enable git for our directory, we use git init command, this will create an empty git repository in the folder. This means now our local folder/directory has converted into working directory.

1. Staging Area

If we want the changes of our files to be tracked we have to use git add filename command. This command will take snapshot of our current file and store it to staging area. And now the git will start to track that file.

* After staging a file, if you modify the file again, those new changes are considered modifications.
* These modification will be happening inside working directory not in staging area, but as we know in working directory changes are not tracked but git is aware of these modification as git will compare the snapshot of the file in staging area with the file in working directory and it will get to know that modifications have happened.
* However, these modification won’t be included in the next commit as they are present in working directory unless you run git add again to stage them for commit.

// to add one file to staging area

git add <filename>

// to add all files of our folder/directory to the staging area

git add .

1. Git Directory

When we commit all the changes which are staged, the snapshot present in staging area will saved in git repository so it means all the changes present in staging area will be permanently saved in git repository so that we can come back to these changes later whenever we want to. It is like creating savepoints on which we can come back later.

git commit -m <Commit Message>

git repository contains snapshot of changes that are commited + date + author name+ message

**What happens when you modify the file again which you have staged**

1. Staging a File

When you stage a file (using git add <file>), Git takes a snapshot of the file at that moment and places it in the staging area. This snapshot is what will be included in the next commit.

2. Modifying the File After Staging

If you modify the file after staging it:

* These new changes are not automatically added to the staging area.
* Git now sees two versions of the file:
  1. The version in the staging area (the snapshot).
  2. The version in the working directory (with the latest changes).

3. Git's Awareness of the Modifications

Although changes in the working directory are not staged, Git is still aware of them. This is because Git compares:

* The staged snapshot (in the staging area) with the file in the working directory.

If differences exist, Git will:

* Report the file as modified when you run git status.

Key Points to Remember

1. The Staging Area and Working Directory Are Separate:
   * Changes must be explicitly staged again using git add if you want them included in the next commit.
2. Tracking Modifications:
   * Git doesn't "track" changes in the working directory by default.
   * Instead, Git constantly compares the working directory's files against the staged snapshot and the last committed version.

**Elaborate Explaination of Stages of a file in Git**

**1. Working Directory (Untracked or Modified State)**

* Definition: The working directory is where you have your actual project files. It's the state of the file on your local machine where you create, edit, or delete files.
* Key Characteristics:
  + Files in the working directory may be:
    - Untracked: Git is not tracking these files yet.
    - Modified: The file is already tracked but has been changed since the last staging or commit.
* Example:
  + After creating a new file:

touch new\_file.txt

git status

Output will show new\_file.txt as untracked.

**2. Staging Area (Staged State)**

* Definition: The staging area (or index) is a place where you prepare files to be included in the next commit. When you stage a file, Git takes a snapshot of the file’s current state and puts it in the staging area.
* Key Characteristics:
  + Files must be explicitly staged using git add.
  + Only files in the staging area will be included in the next commit.
* Example:
  + To stage a file:

git add new\_file.txt

git status

Output will show new\_file.txt as staged, ready to be committed.

**3. Repository (Committed State)**

* Definition: The repository contains the committed history of the project. When you commit files, their state is saved permanently in the local repository.
* Key Characteristics:
  + Once a file is committed, its state becomes part of the repository's history.
  + Tracked files remain part of the repository even after modification, but modifications need to be staged and committed again to update the repository.
* Example:
  + To commit a staged file:

git commit -m "Initial commit"

git status

Output will show the working directory as clean, meaning there are no uncommitted changes.

**Commands for Git Bash**

1. pwd (print working directory) used to see in which folder we are
2. cd pathOfFolder (change directory) to move from one folder to

another. For ex- cd D:\Probation\Git\NewFolder

1. ls to list all the file and folders you are in.

ls -a to see hidden folders

ls -al ls : Lists the contents of a directory.

a : Shows all files, including hidden files (those starting with a .).

l : Displays detailed information in a long format, including permissions, number

of links owner, group, file size, and timestamp.

For ex – ( . currently in which folder we are present

.. parent folder of current folder

and hidden folder/files

and non hidden folder/files.)

1. q to exist the command
2. Exit Vim in Git Bash

To Save and Exit (Complete the Revert):

* 1. Press the Esc key to ensure you're in command mode.
  2. Type :wq (write and quit) and press Enter.

To Exit Without Saving (Abort the Revert):

* Press the Esc key to enter command mode.
* Type :q! (quit without saving) and press Enter.

1. cls/clear to clear the terminal screen

1. git init to add git to that folder. (it will convert our folder into git

repository)

1. git add filename.ext it will move the modification done in working directory to staging

area.

For example git add APP.java

1. git add . to move all the changes to staging area from working directory in one go.
2. git status (shows current state of our git repo) name of the files modified, staged, and

untracked

git diff it shows modification happened in working directory

1. git commit -m “write your message” it will save our changes permanently in our repository.
2. git commit -am “message” (same like git add and git commit) but in one command,

a means send modification to staging area and then

commit changes

1. git restore fileName it will discard the modification happened in working

directory and take back to file to latest commit.

1. git restore --staged fileName it will discard all the changes in staging area.

1. git checkout commitId move back to that particular commit changes.
2. git pull after cloning, to pull commits of remote repo to our local

repo

**REVIEW A REPO HISTORY**

git log show all the commit history with commit hashvalue, name of author(who did

the commit), date of commit and message given at the time of commit

git log -n here n represents the number of commits you want to see

show n number of latest commits.

For ex git log -5 …… it show 5 latest commits

git log -p show all the commit history and the acutal changes in files

combination of git log + git diff

git log --oneline show all the commit history but only commit hashvalue and commit

message

use case- if we want to find any commit fastly and we remember its

message use this command.

git log --stat show all the commit history and number of changes in each file in

each commit

git show commitId show particular commit history and all the changes done in that

particular commit

**.gitignore file**

* .gitignore file 🡪 we mention those file name inside the .gitignore file which we want Git should not track.
* We can mention file name(for ex documents.txt) git will ignore document.txt file, we can mention path also (for ex – assets/images/\*.png) git will ignore all .png files inside images.
* And we commit .gitignore file as we want to keep this file so we know which

files we are not tracking.

**Two ways to make Git Repository**

1. Creating our own local git repo

Make a folder in your system and add git to that folder and that folder will be converted to git repository..

**How to do this**

Make a folder into your system and go inside that folder open git bash, use git init

command to initialise that folder with git

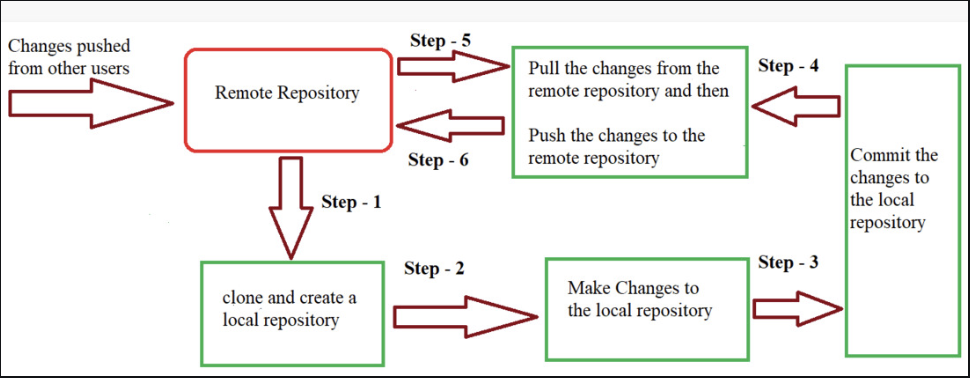
1. Directly clone repository from Github

Clone – means bringing git repository which was stored in remote(cloud) into your local system.

**How to do this**

Open GitBash(wherever you want to bring git repo) type git clone pasteURL

command. For ex git clone https://github.com/Pathfinding-Visualizer.git

****

• In Step – 1 We first clone any of the code residing in the remote repository or make our own local repository.

• In Step-2 we edit the files in our local repository and make the necessary changes in it.

• In Step-3 we commit our changes by first adding them to our staging area and committing them with a commit message.

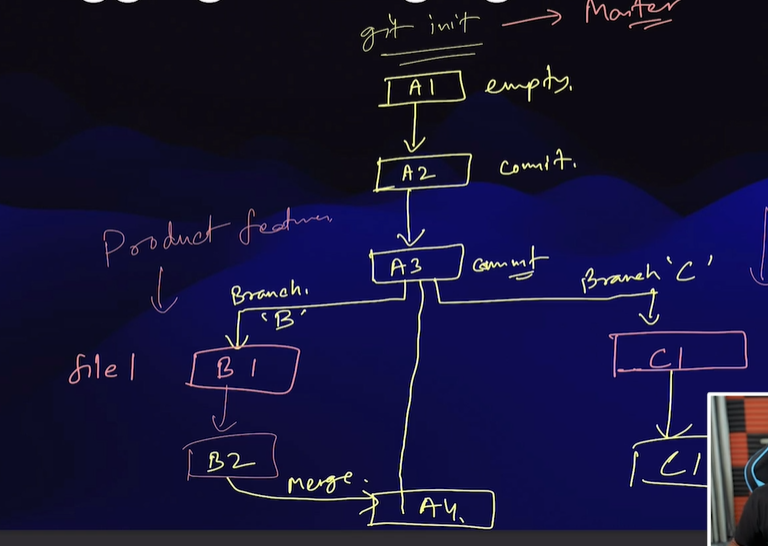
• In Step – 4 and Step-5 we first check whether there are any of the changes done in the remote repository by some other users and we first pull that changes.

• In Step – 6 we push our changes to the remote repository.

**Notes**

* + 1. HEAD will always point to latest commit.

**branching, tagging & merging**

****

Intially when we initialize a git repository at that time we have only one branch called master and all the commits we do, by default they will be carried on master branch only.

But good practice is, we need to create a custom branch and do all commits in that custom branch, and later when we are sure about all the commits in custom branch then we merge custom branch in main branch because master branch should always be in maintable and stable state.

But it may happen that in a big project, multiple teams are working and each team is working on separate feature so at that time we create custom branch.

And later when any team is ready with its feature it will merge its branch to master branch.

This way master branch will always be the maintainable and working state and each team can work separately in same project without affecting other team.

custom branch can be created from any branch (main branch also and any other custom branch also), so if we create custom branch while we are in main branch,(it will be sub-branch of main branch) but if you create new branch from custom branch(it will be sub-branch of that custom branch).

For merging custom branch to main branch, we first come to main branch and then merge the custom branch.

Deletion of custom branch can be done from any branch (main branch also and custom branch also).

**Merge Conflict**

A merge conflict means when Git is unable to merge two branches automatically.

And this happens when in both the branches, the code is present in same line and in same file and we are trying to merge both the branches so git will get confused that which changes it has to keep because in one line only branch code will come, so merge conflict will come.

**Resolving merge conflict**

* 1. Resolve the conflict by manually doing changes in code keep or remove whatever you want from the file.
  2. Then tell git conflict has been resolved by adding modification to staging area(git add command) and then commit the changes.(git commit command)

**branch commands**

1. git branch give you names of all the branch present in git repository
2. git branch branchName create a new branch
3. git checkout branchName switch to that particular branch
4. git checkout -b branchName create a new branch and switch to that new branch also
5. git branch -d branchName delete that particular branch
6. git merge branchName merging custom branch to main branch

**tagging**

tagging means giving tag to a particular commit.

useCase – if I want to tell that this particular commit is my Beta release so I can give that particular commit a tag called Beta release. (this can be seen using git log command)

**tagging Command**

1. git tag -a nameOfTag commitId -m “message” give tag to that particular

commit

1. git tag -d nameOfTag delete that particular tag from

the commit

**stash stack**

git stash area is a place where we move our modification done in local repo, so that we can pull the latest commits from remote repo.

First Secenario

Suppose we have clone a remote repo in our local system and after that we did modification in our local repo, and after we cloned, someone has pushed some changes in remote repo, in the same line and in same file where we also have done modification in local repo.Now remote repo is updated and we are working in old code base.

Now when we pull changes, changes of remote repo will not be pulled as we have modification in same line and in same file, in our local repo also and in remote repo also. So it will give error and not pull any change.

So that is why you first move your modification to stash area, so now there will no modification in our working directory, now git can easily pull the changes of remote repo to your local repo.

And after pulling changes of remote repo, then move your modification from git stash area to your local repo. Now merge conflict will come because your modification are also on the same line where new changes just merged.So resolve merge conflict and commit the changes.

Q- Why merge conflict do not come here and give error and aborts the merge?

Merge conflict only occur when we are merging two committed changes from two different branch or repositories.

Here modification done in local repo is not tracked by git(as they are happening in working directory) but Git is only aware of the modification (by comparing the last snapshot of file in staging area or git repo with the working directory). These modification are not yet become part of git repo,so git can’t compare the modification of local repo with changes coming from remote that why merge conflict didn’t came.

But git is aware that applying the changes from remote branch would override the modification, which results in data loss. So to protect our data, git stops pull operation.

Second Secenario

we cloned a remote repo, and after that done some commits in our local repo, but after we cloned, in remote also someone has pushed some changes in the same file and in same line where we have done commits in local repo, now when we pull the changes, merge conflict will come

So resolve merge conflict and commit the changes.

**stash commands**

git stash move your modification to stash stack

git stash list list the modification stored in stash stack

git stash apply move your modification back to working directory

**Commands for pushing your code in your remote repo**

git remote add origin remoteUrl

git push -u origin nameOfBranch

**Undoing Commits**

* 1. **Reverting a commit**
* reverting a commit means it removes the changes of that particular commit(whose commitId we have given) and create a new commit.

git revert commitId revert a commit

* 1. **Reset a commit**

(any one)

git reset --soft/mixed/hard commitId move the branch pointer(HEAD) to a particular

commit(whose commitId you have given) and undo

the changes of removed commits.

soft/mixed/hard

soft means it will move all the changes of removed commits to staging area.

mixed means it will move all the changes of removed commits to working directory.

hard means it will deletes all the changes of removed commits.

For example

Assume your commit history is:

A -> B -> C -> D (HEAD)

Reset to Commit B:

* git reset --soft B:
  + History becomes: A -> B (HEAD)
  + Changes from C and D are staged.
* git reset --mixed B:
  + History becomes: A -> B (HEAD)
  + Changes from C and D are unstaged but still in the working directory.
* git reset --hard B:
  + History becomes: A -> B (HEAD)
  + Changes from C and D are deleted completely.
  1. **Amending a commit**

allows you to modify the most recent commit. It can update the commit message, include additional changes, or both

git commit --amend

1. Amend the Commit Message

If you only need to correct the last commit's message

Run the command:

git commit --amend

The editor (e.g., Vim, Nano) opens. Modify the message and save.

The new message replaces the old one, but the commit content remains unchanged.

1. Amend the Content of the Commit

If you forgot to include changes in the last commit you can include these changes in

previous commit.

Make and stage the changes:

git add <file>

Amend the commit:

git commit --amend

Modify the commit message in the editor or leave it unchanged.