Git Lecture 02 (Git Starters) Notes by Milind Mishra

✓ git init - powers your folder to be managed by git, & initializes a new repository. It also creates a (.git) folder that comprises all features that help with version management.

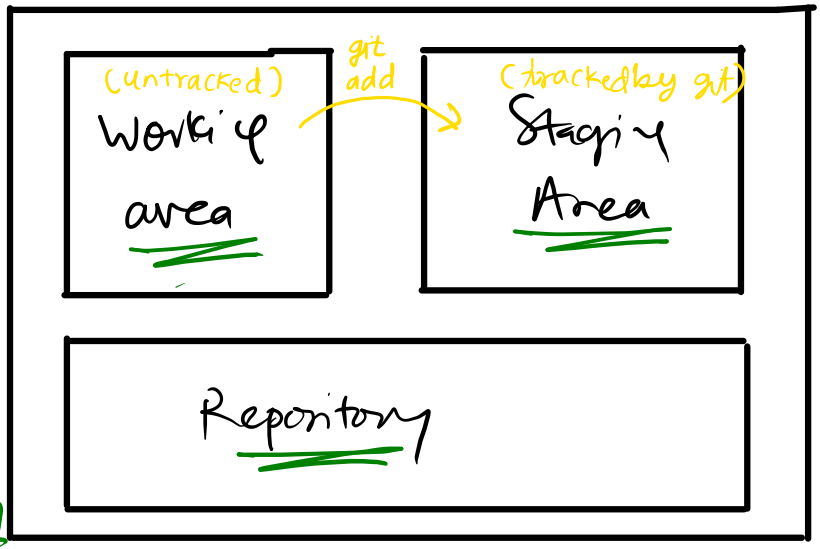
Repository (git Repository) → enabled w/ git features

→ ls -a

.git

✓ git status → spits branch info, tracked/untracked files... etc.

3 areas where files changing / changes can reside



any changes in files are usually untracked by git.

i.e. on $ git status ↓ shows untracked

"VCS" → makes sure we can manage diff. versions of software

To let git manage the code versions.

Areas

1. Working Area :— There can be a bunch of files that aren't handled by git.

i.e changes done or to be done aren't managed by git.

file @ work is untracked & is not in the staging area.

any new file you make is in working Area & is untracked.

Staging Area → Lets us know what files are supposed to be a part of next version.

↓

place where, git knows what files / file changes will be in next ver.

diff(s).

by doing $ git add → <file> * ext.

→ moves files from W.A to S.A

↓

Staging.

to unstage git rm --cached <file>

by doing $ git add . → add all untracked files to staging Area.

Repository Area? → Area actually contains details of all previous versions.

files here → git manages & knows file history.

To, Register a version → git add.

Move from staging area to repository Area → "commit."

commit → It is a particular version of the project. It captures a snapshot of projects staged changes & creates a version out of it.

$ git commit → registers staging changes to a commit.

Vi → Vim editor

i → insert mode

esc, :wq → save & exit.

$ git log → spits all commits of repo

every commit is a version & has a # hash.

* to exit out of git logs → press Q.

{ - lines. / + lines. } → info on diff(s).

$ git restore <file> → discards all changes.

→ deletes every thing @ staging area that is yet not commited

→ removes all files changes that are to be committed.

restores last clean version of project.

* use full if there are breaky changes in staging area.

git add <filename> → whatever changes made goes to staging area.

To, bring file back to working area → [git restore --staged <filename>]
from staging area.

To, revert it alltogether → git restore <filename>

→ git restore <filename> – it removes all files changes from staging area to be comitted. This can be use full, if we break something & now don't want it. So, insted of deleting every change line by line, we can restore it or you can say restore last clean version of file.

→ git restore --staged <filename> → removes file changes from working area to Staging area

NOTE: only works if changes are at your "staging area".

* git restore → moves from working copy to absolutely clean copy.

* git restore --staged → moves file changes from staging area to working area.

"unstage" basically.

QUICK REVISION:

How do we stage changes? You may ask... [git add <filename>] ✓

WORKING AREA

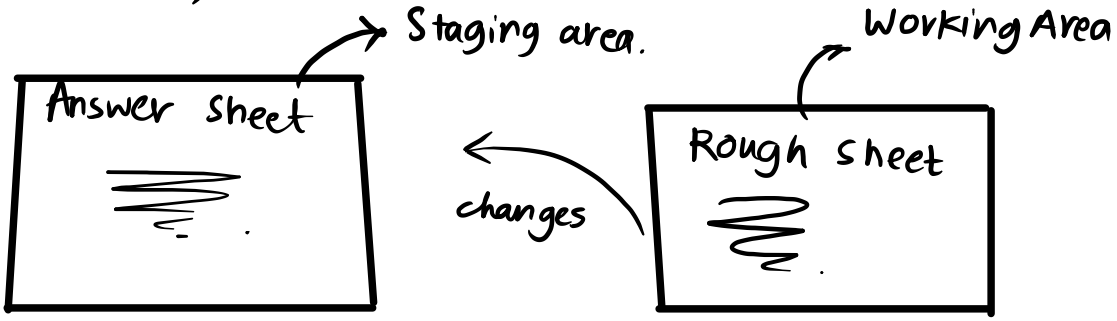
to be staged changes...

V/S

STAGING AREA.

Part of next commit...

eg: Exam Scenario,



* Difference b/w "git rm" and "git restore"

"git rm" → moves files to "untracked state"

"git restore" → to just move changes to Working area -or- Staging area.

git rm --cached <filename>

* git log → spits details of commits & Commit id(s).

* git diff <commit id 1> <commit id2> → {handles, multiple states of file changes.}

= } deleted content
+ } added —"—
b/w commits.

"devhints.io/vim" → vim cheatsheet

Commit messages:

git commit -m "Message content"

for just git commit → opens vim to add commit messages.

GitHub → "+" icon → new repository → name → public/private → create.
(remote repository created!)
-or- (repo.new)

Now to connect the remote repository to our local git repository.

$ git remote add origin <URL> ✓ → Links two repository
↓
local + remote!

syntax breakdown:

`git remote add origin <URL>`

- `git remote` — connects two repository & a starting command for all local to remote operations. (spits the remote conn^n name.)
- `add` — adds remote repository
- `origin` — name of the specific remote connection can be anything o1, o2 ...etc.
- `<URL>` — remote VCS provider URL. (must be a valid git link.)

* We can also remove the remote connections to our local repository by:

$ git remote rm origin

remote conn^n name.

to upload changes to remote branch from our local repository

$ git push origin master → uploads to github repository

↳ default branch.

* All changes / commits / branches can be viewed @ Github.

+ 5 commits

(spits all commits.

git remote → lists down all remote connection names.

Remote repository → helps you link two git repositories for uploading & downloading changes from each otherwise.

git remote add <name of remote> <link of remote> — helps add new link of remote repo. and gives name to it.

git remote rm <name of remote> — removes remote connection.

git remote rename <old remote name> <new remote name> — renames the remote conn^n.

NOTE: the name of remote is used to establish connection b/w repo(s). ✓

You can make changes @ GitHub itself, to sync them to local → "git pull."

To add multiple file changes

$ git add <file1> <file2> ...

-or-

$ git add .

↖ adds all incomming changes in files to staging

. → refrences to current folder/ all files @ current dir.

.. → parent dir. refrence.

$ git pull origin master — downloads all latest changes from branch of mentioned remote into your local repo.

<remote name>

<branch name>

* Recommended Practice:

Flow:

- make changes
- git add <file>
- git commit
- git pull
- git push

Merge Conflicts: ⚠

When there are existing local changes in your repo & remote incurrs same file changes & when we pull, we suffer Merge conflicts!

pull → auto merges local + remote.

To resolve them → accept inc. change
decline
accept both
use merge editor @ VS Code.

push again w/ -m "resolved" ✓

* happens, when working on same file w/ multiple contributors!