Git:

- It is very popular version control system that is used to manage projects.
- A version control system is software that is used to track changes made to our project and it stores these modifications as different versions of our project.
- Git is a version control system used to track changes in computer files. it is a tool to manage your code & file history while coordinating work remotely on those files with others.
- git is simple tool which can install in local system, which is acting as a local repository.
- A major advantage of using Git, or any other version control system, is that we can easily compare the current version of our project with previous versions and rectify errors. We can also roll back to the previous versions.
- It is a distributed version control system so that anyone who clones our repository will have access to the entire commit history.

How does Git work?

- Git manages the different versions by maintaining a series of snapshots of the project. Whenever we want to commit or store a change permanently a new snapshot of our current version is taken and stored by Git.

GitHub:

- GitHub is a hosting service for git repositories.
- Git is the tool, while GitHub is the service to use git
- GitHub is service which is acting as a remote repository
- GitHub is a Git Repository hosting service that is owned by the tech giant Microsoft. We can upload our local Git repository to GitHub and access it anywhere.

Disadvantages:

- Git provides mediocre support for binary files like images and multiple changes to such files make Git a bit cumbersome to use.
- Git has a steep learning curve and one needs to put in the time and effort to master it.
- Git is somewhat slower on the Windows platform as compared to other platforms.

Git Architecture/Workflow

Working with Git consists of moving between three stages. Let's take a look at these three stages.

Working Directory ---> Staging ---> local repo ---> Remote Repo

Working Directory:

The place where we can make changes to our project like adding new files and deleting or modifying the existing files. Whenever we make any change to our files we move back to this stage.

`Untracked files`

Staging Directory

Working directory or workspace we cannot directly send them to the local repository that means we cannot directly commit them into local repository so it should go through staging so staging is another layer which is basically a virtual layer between our workspace and your git local repository tracked files

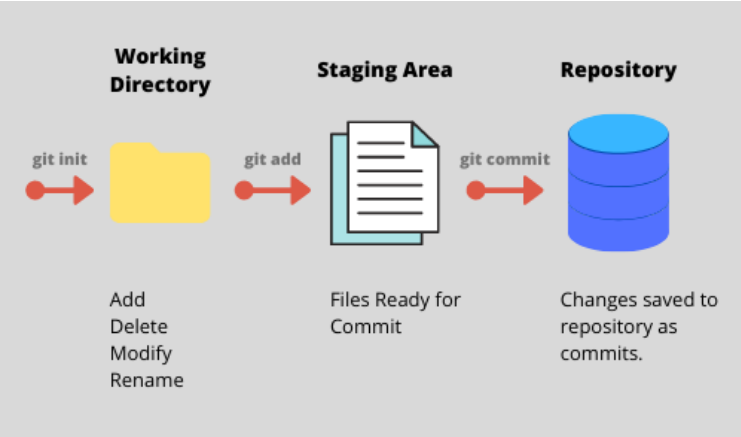
$ git ls-files --stage (show staging area files)

Stage Changes: After making modifications, you add these changes to the staging area using **git add <file> or git add .** (to add all changes).

The place where we can list the changes that we made to our files. The main aim of this stage is to segregate and organize the files that we have to commit. Changes can be made to multiple files in the working directory but we can add only those files in the staging area that we have to commit. This helps in keeping the project versions simple and easy to understand

Git Local repository

Once your files are there in the local repository, we can call them as a committed file or committed files The place where Git permanently stores or "commits" these changes as different versions of our project. All the snapshots of the project versions are stored here along with some additional information.



Git Init

Git init is going to be the very first command that we will always use when working with Git. It is used to initialize a new Git Repository and tell Git to track any changes that we make in that repository. We only need to do this once while creating a new Git Repository. To convert an existing directory into a Git repository we need to move to that directory and just run the git init command directly. Let's take a look at how to use the git init command.

```
$ git init
```

Git Clone

To clone a remote repository to our local system, we need the URL to that remote repository. Let's take a look at how to use git clone command.

```
$ Git clone "github url"
$ git clone <remote-repository-url> <directory-name>
```

The example shown below will clone only the last 4 commits that were made to the project.

```
$ git clone -depth=4 <remote-repository-url>
```

Git Add

The next command that we will be using after initializing a Git repository is **git add** command. The *git init* command lets us create and edit files in the working directory but it is the *git add* command that allows Git to **track** a particular file and check for any modifications to it.

```
$ git add 1.txt (adding file from workspace to staging area)
$ git add .*  (adding all . files which are started with. from workspace to staging area)
```

Git status

it will show files status which are available in workspace

```
$ git status
```

Git Commit

Git does this by capturing a snapshot of the state of the files present in the Staging Area and storing it permanently in the repository as a new version of the project.

At any point in time, we can move back to any of these versions that we committed.

```
$ git commit -m "this is my first commit" (commit to local repo)
$ git commit -a -m "both the files are modified"
```

Git .gitignore file

The .gitignore file is a text document that tells Git what files it should ignore. We don't always want Git to keep a track of every file on our repository and we can ask Git to ignore the changes that we make to our files with the help of a special .gitignore file.
By ignoring we mean that Git should not track any changes made to these files.

Git Log

Git is used to save different versions of our project in the form of snapshots stored in commits. The Git log command is used to view the commit history of our repository.

```
$ git log (it will show how many commits you have done earlier)
$ git log -1 (it show the latest commit)
$ git log --oneline (it will show in one shot)
$ git log –-stat (It will tell us how many lines were added and deleted to the files.)
$ git log --graph (command to view the commit history in the form of a graph)
$ git log --after="YYYY-MM-DD" (To view commits made after a certain date:)
$ git log --before="YYYY-MM-DD"
$ git log --after="YYYY-MM-DD" --before="YYYY-MM-DD”(To view commits made between two dates:)
$ git log --author="author-name"() commits made by a certain author by using the --author option.
$ git log --grep="commit-message” (commits that have the matching words in their commit message)
```

Git Config

- Git Config (short for configuration) is a command that helps us in setting various fields in Git like our name,email address, the default editor that should be used, etc. This is often done after installing Git on our system but these fields can also be changed at some other time in the future.

- The three levels of configuration are local, global, and system.

Local Git Config

- We can change the configuration at the local level i.e. the current Git Repository on which we are working.
- To change any fields at the local level we use the --local option along with the Git Config command. These settings reside in the .git directory by the name of config. We can view these configured values using the following command.

```
$ git config --list --local
$ git config --local user.name "acmarpu"
$ git config --local  user.email "acmarpu@gmail.com"
```

Global Git Config

- We can set some global settings for our Git Repositories. These configurations will apply to all the git repositories for a particular user of a system.
- We use the --global option to alter these configurations. All these settings are stored in a .gitconfig file in the home directory

```
$ git config --global  user.email "acmarpu@gmail.com" (configuration command)
$ git config --global  user.name "acmarpu" (configuration command)
$ git config --list --global
```

System Git Config

We can also configure the Git Repositories at the system level. These settings will apply to all the Git Repositories of all users that share a system. We use the --system option to change these values. These settings are stored in the <git-install-root>\mingw64\etc\gitconfig file.

```
$ git config --list --system
```

---------------   Not Updated, it Will be update soon ----------------------

---

git diff -- (to compare the file contents between working directory or staging or between working directory
or local repository or between local repository remote repository we can use git get diff command basically used for comparing the files)

Ex1:

```
git diff index.txt
diff --git a/1.txt b/1.txt
index 6cf76c1..59c29a5 100644
--- a/1.txt
+++ b/1.txt
@@ -1,3 +1,4 @@


-today modified
\ No newline at end of file
+today modified
+2nd tiem
```

a/1.txt represent source (staging area)
b/1.txt represent destination (working dir)

6cf76c1 --> Hash of the fiel content from source/staging
59c29a5 --> hash of file content from destination/workspace
100644 git file mode
   100 represent type of the file
   644 file permissions re-r-r
     4 -r
     2 -w
     1 -e
--- a/1.txt source file missing some lines
+++ b/1.txt new lines added in destination file (working dir)

if anyline prefixed with space means it is unchanged
if anyline prefixed with +means it si added in destination copy
if anyline prefixed with -means it is removed from destination copy

Ex2: To see the difference in file Content between Working Directory and last commit

*Last commit refred using HEAD
git diff HEAD index.txt

Ex3: To see the difference in file content between staged copy and last commit
git diff --staged HEAD index.txt

Ex4 : To see the diffrence in file content between specific commit working directory copy
git log --oneline
git diff d0ff720 index.txt

Ex5 : to see the diffrence in file content between specific commit and staging are copy
git diff --staged d0ff720

Ex6 : To see the diffrence in file content between 2 specified commits
git diff d0ff720 c0ff720 index.txt

Ex7 : To see the diffrences in content between 2 branches
Master
test
git diff master test -- it shows all differences between master branch and test branch

Ex8 : to see the diffrences in content between Local and remote Repositores
git diff master origin/master

git rm remove files from both staging & working directory
git rm file.txt
git rm -r .

Remove files from only from staging
git rm --cached file2.txt

Git Checkout command:-
git checkout -- 1.txt (undo operations)
We can use checkout command to discard unstaged changes in the tracked files of working directory
1) only for working directory
2)to discard unstaged changes (The changes which are not added to staging area)
3) in the tracked files (The files which are already added to staging area/commit)

---
git reset 1.txt (undo operations from git stageing area)
To remove changes from staging area
To undo commits at repository level
git reset 1.txt (from git stageing area)

git reset <mode> <commit id>
Move the HEAD to the specified commit, and all remaining recent commits will be removed
mode will decide whether these changes are going to remove from staging area and working director or not

--mixed
--soft
--hard
1) git reset with --mixed mode
    it is the default mode
    to discard commit in the local repository and to discard changes in staging area.
    it will not touch working directory
2) git reset with --soft mode
    it is exactly same as --mixed option, but changes are available in working directory as well as staging
    it wont touch staging area and working directory
    as changes already present in staging area, just we have to use commit to revert back

3) git reset with --hard mode
    it is exactly same as --mixed except that changes will be removed from everywhere (local repository,staging area, working directory)
        it is impossible to revert back and hence while using hard reset we have to take special care

---

Branching

Git Branching (only show active branches)
git branching (to view available branching)
git branch --list
git branch -a (show all brnches)
git branch branchname (to create new branch)
git checkout (switchto from one branch to another)

```
git checkout -b baranchname (creating & switching to branch)
git push --set-upstream origin <branch name> (push created branch)
-------------------------------------------------------------------------------
-----------------------
Merging :-
-----------------
Fast-Forward Merge
Three way merge

How To Resolve Merge Conflicts :-
git merge feature (master to feature branch )

Git Merge:-
will Keep all commits history of the feature branch and move them into the master
branch will add extra dummy commit

Squash Merge:-
Will Group all feature branch commits into one commit then append it in the front
of the master branch will add extra dummy commit.

Rebase Merge:-
Will append all commits history of the feature branch in the front of the master
branch Will NOT add extra dummy commit

Semi Linear Merge:-
it's a mix of rebase and a merge. first the commits in the pull request are
rebased on top of the master branch.
Then those rebased pull request are merged into master branch

Chery pick
Select individual commit to copy
git cherry-pick <commitID>
-------------------------------------------------------------------------------
--------------------

git push origin main
git pull origin master
git push --set-upstream origin brnachname
git push origin

-------------------------------------------------------------------------------
--------------------

push your local repo into remote
Create a local repo and remote repo
```

```
copy url and add into local
git remote add origin url
git push --set-upstream origin master
git push origin main
git push origin master:main

#########################################################
git remote add origin <https:>
git push -u origin --all
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

git push --set-upstream origin terraformbranch

$ git remote -v
```

URL
Set environment variables // GitHub Actions
https://www.youtube.com/watch?v=hAAAQL03f9E&list=PLy4OcwImJzBKzWWb9K_WB3QzaxoiGmxyo&index=7