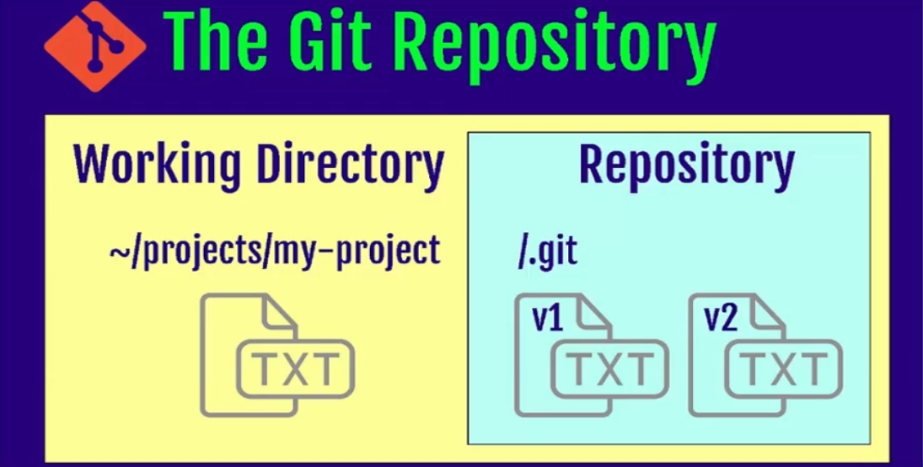
**Git & GitHub**

1. **Goals & Objectives**

* Understand the core concept of GIT
* Master in main Git commands
* Compare differences with ease
* Branching and merging in GIT
* Exposed to GITHUB
* Be pro in GIT
* Bonus Materials.

1. **Git Terminologies:**

**Repository** : Git keeps everything on the repository on user’s local machine. The repository contains files, history, config managed by git.

Three Stages of GIT:

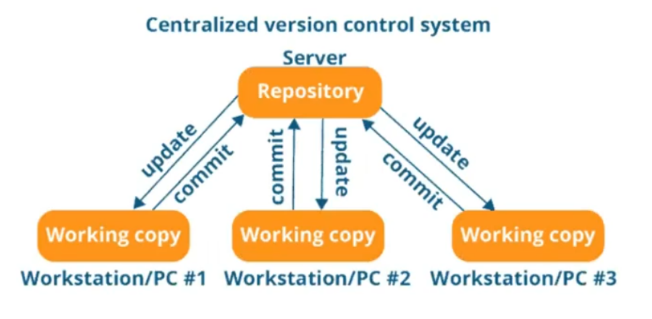
* Working directory: This is where you make changes to your files. It’s your local folder where you edit, create, or delete files.

Staging Area- pre Commit Holding Area: After editing a file, you decide you want to save these changes. You use git add to move your changes to the staging area. It’s like putting items in a shopping cart before checking out.

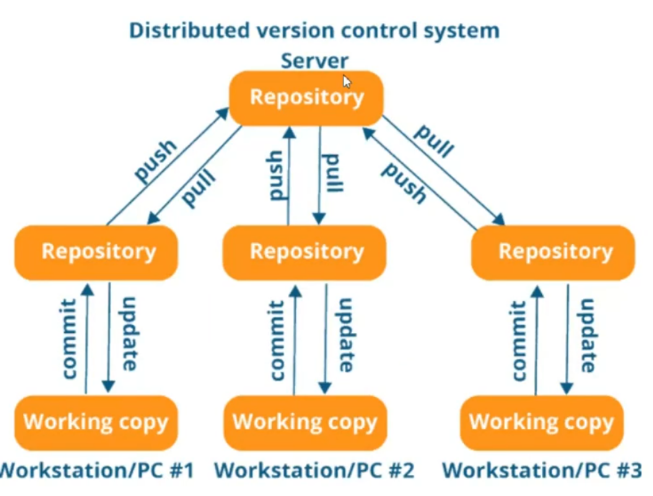
Commit – GIT Repository(History): Once your changes are staged, you run git commit. This records those changes, along with a message describing what you did. It’s like taking a snapshot of your work that you can look back on later.

* Remote Repository(GITHUB): After committing your changes, you push them to GitHub using git push. This uploads your changes from your local repository to the remote repository, making them available to others.
* Master Branch: The master branch is the main line of development in your project. It’s where the stable, final version of your project lives.

**Version control System** :It is the management of changes to documents, computer programs, large websites and other collection of information. Two categories:

**Centralized VCS(CVCS):** It uses a central server to store all files. It works on a single repository to which users can directly access a central server. Central server could be local or remote machine directly connected to each of the programmer workstation.

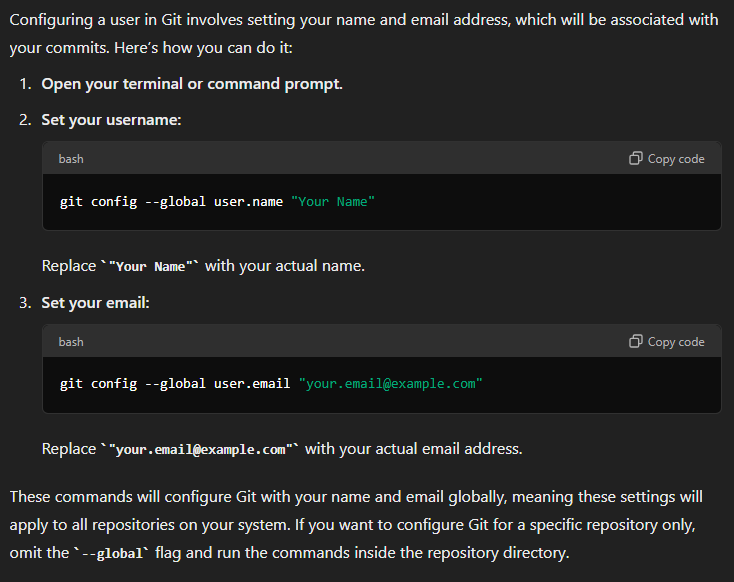
Drawback: Since everything is centralized, in any case of central server getting crashed, will result in losing entire data.

**Distributed VCS(DVCS):**In DVCS, every contributor has a local copy or clone of the main repository. Developer can update their local repositories with new data from the central server by an operation called pull and affect changes to the main repository by an operation called push from their local repository.

**Local Copy**: When you start working on a project, you make a complete copy (or "clone") of the main project on your own computer. This means you have the whole project and its history, so you can work independently.

**Pull**: If someone else has made changes to the project, you can update your local copy by "pulling" the latest changes from the central server (like GitHub). This keeps your copy up to date with everyone else's work.

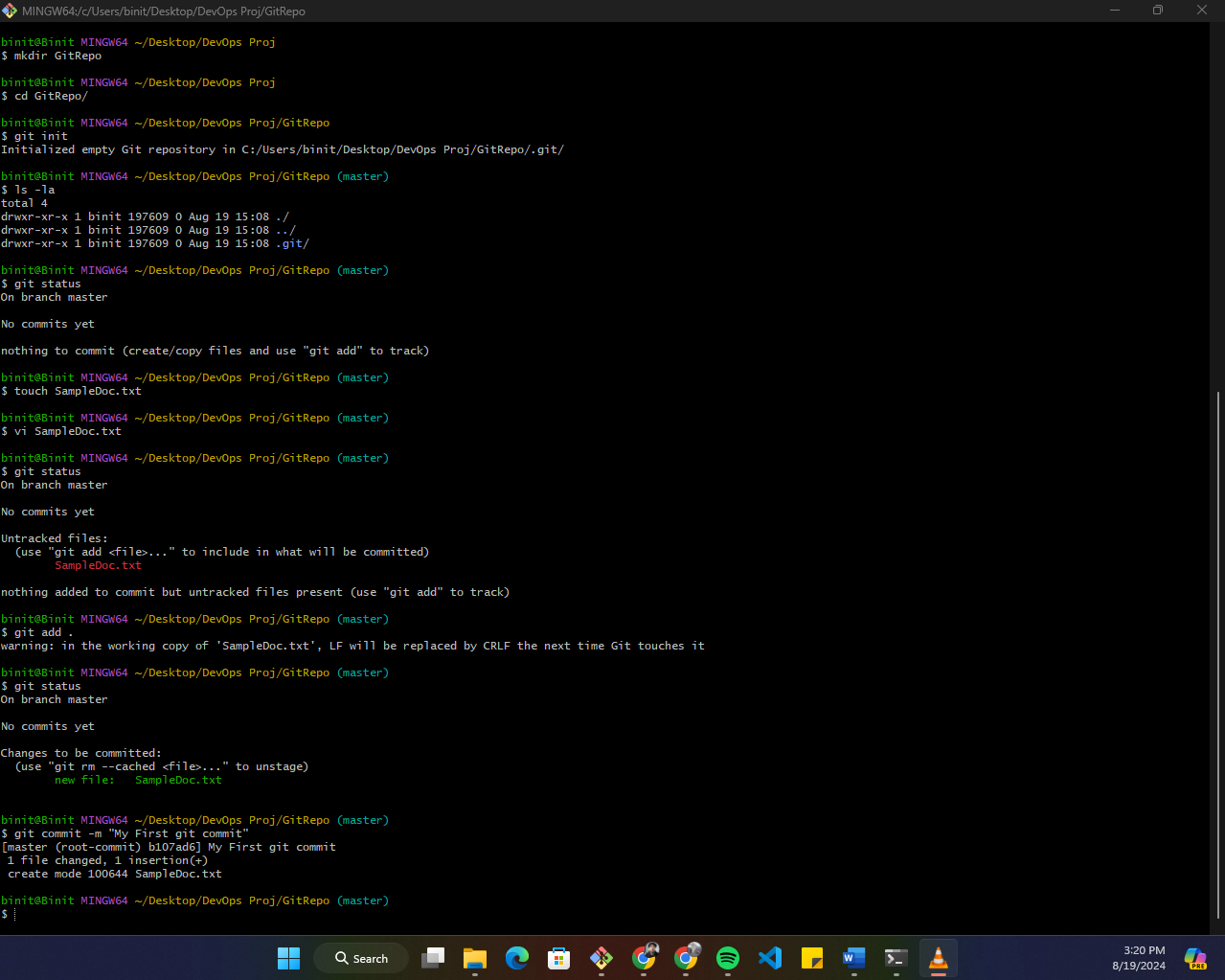
**Push**: After you make your own changes and commit them, you can "push" these changes from your local copy back to the main repository on the central server. This updates the central repository with your work, making it available to everyone else.

1. **Configure User in GIT**

**A screenshot of a computer

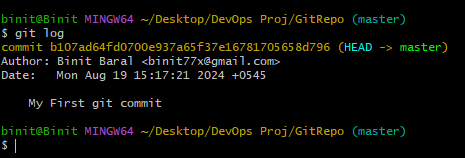
Description automatically generated**

1. **First Project in GIT**

**git status**: shows the current state of your working directory and staging area, including which files are staged for commit, which are not tracked, and which have been modified but not yet staged.

**git add .** **:**  stages all the changes (new, modified, or deleted files) in the current directory and its subdirectories, preparing them to be committed.

**git commit -m "message**" : creates a new commit with the specified commit message "message", documenting changes made to the repository.



**git log**: displays a chronological list of commits in the current branch, showing commit hashes, authors, dates, and commit messages. The Git ID, or commit hash, is a unique identifier for each commit in a Git repository. It allows you to reference, compare, and retrieve specific commits, facilitating operations like viewing past changes, rolling back to previous states, and merging changes.

A screenshot of a computer screen

Description automatically generated

**git log --author="Binit Baral"** : filters and displays commit history authored by "Binit Baral”

1. **Understanding of GIT workflow**

**A blue rectangles with white text

Description automatically generated**

**Working Area**: Where you edit files.

**Staging Area**: Where you prepare files for commit.

**Repository**: Where committed changes are stored.

1. **Use if Text editor in devops**

**Sublime Text** is used for writing and editing code, scripts, and configuration files. In DevOps, you often need to work with various files like YAML, JSON, shell scripts, Dockerfiles, and Terraform configurations, all of which can be edited using Sublime Text.

**Git** is a version control system, and Sublime Text can be integrated with Git through plugins, allowing you to perform Git operations directly from the editor.

You can **stage, commit, push, and pull changes**, view diffs, and manage branches, all from within Sublime Text. This integration streamlines the process of code management and collaboration.

To configure the default text editor that git uses (e.g. for commit messages), you can use the following command

**Sublime text:**

**git config --global core.editor "subl -n -w"**

**VIM:**

**git config --global core.editor "vim"**

**Configuring notepad++ as a git text editor in windows**

1. Download and install notepad++ from the browser
2. Add the path to environment variables for the system use.
3. Open git bash and create .bash\_profile by following command:

**Notepad++ .bash\_profile** (this command will ask you to create new file if .bash\_profile doesn’t exists)

**What the Command Does:**

* **Opening the File:** When you run notepad++ .bash\_profile, it tells Git Bash to open the .bash\_profile file using Notepad++.
* **Creating the File (if it doesn't exist):** If .bash\_profile doesn't exist yet, Notepad++ will ask if you want to create a new file with that name. You can create it by clicking "Yes."

1. Add new line inside notepad++ to add an alias.

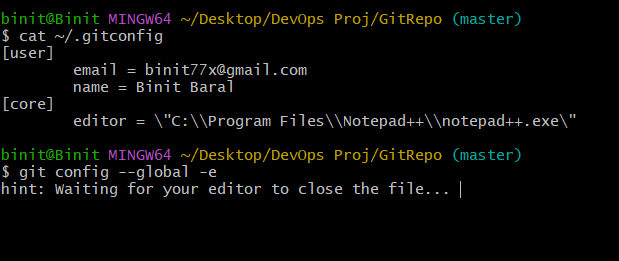
**Alias npp=’notepad++.exe -multicast -nosession’** then save and close it.

**The line alias npp='notepad++.exe -multiInst -nosession' is a shortcut you create to make it easier to open files in Notepad++ from Git Bash**.

* **alias npp=:** This sets up a shortcut (or alias) in Git Bash. After creating this alias, you can type npp instead of the full command to open Notepad++.
* **'notepad++.exe -multiInst -nosession':** This is the actual command that runs when you type npp.
  + **notepad++.exe:** This tells Git Bash to run the Notepad++ program.
  + **-multiInst:** This option allows you to open multiple instances of Notepad++. For example, if you open a file with npp, it won't close your previous Notepad++ window; instead, it opens a new one.
  + **-nosession:** This option makes sure Notepad++ doesn't load previous files when you open a new instance. It opens only the file you want to edit right now.

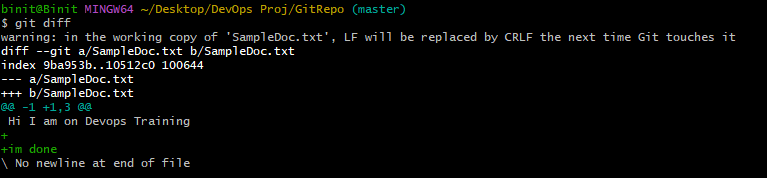
1. Restart gitbash
2. Then cinfigure notepad++ with git.

**Git config –global core.editor “\”<path\_of\_your\_notepad++>\notepad++.exe\””**

**This command is used to set notepad++ as the default text editor for Git.**

use this command to verify wheather notepad++ is configured or not

1. **Verify Changes in Git**



**git diff**: shows the differences between the working directory and the staging area or between commits, highlighting changes made to files.

**a/SampleDoc.txt** represents the file before changes, and **b/SampleDoc.txt** represents the file after changes.

The index **9ba953b..10512c0** line refers to the specific commits or states of the file, represented by the unique hash values (9ba953b and 10512c0).

But.. if I create a new file it will not show in git diff because it is staged 0 times

1. **Git Diff vs Git Diff - -Staged**

**Git diff**: Shows the differences between your working directory (what you're currently editing) and the staging area (where files are prepared before committing).

**Git Diff - -Staged**: Shows the differences between the files in the staging area and the last committed version of those files.

1. **Delete File in GIT**

**Git rm <file name> :** When you run git rm test.txt, Git will delete the file test.txt from your working directory (i.e., your local file system). This means the file will no longer exist in your project folder. Git automatically stages this deletion, which means it marks the removal of test.txt for the next commit. The file is added to the staging area but as a removal.

To restore the file back to the working directory and unstage its deletion, you can use:

**git restore test.txt**

If you want to restore the file in your working directory but keep the deletion staged, you can run:

**git restore --staged test.txt**

If the commit has already been made, you can use git revert to create a new commit that undoes the file deletion:

**git revert <commit-hash>** (commit hash means unique commit id)

**git reset HEAD <filename>:** The command git reset HEAD <filename> is used to unstage a file that you have previously added to the staging area using git add.

**Git reset HEAD~ :** The command git reset HEAD~ is used to undo the last commit in your local Git repository, while keeping the changes you made in the working directory.

**git diff HEAD:** This command shows the differences between the current state of the files in your working directory and the last commit (HEAD). It includes both staged and unstaged changes.

In Git, **HEAD** is a reference to the latest commit in the currently checked-out branch. It acts as a pointer that tells you where you are in the commit history.

To compare the staging area (also known as the index) with the latest commit in the Git repository (referred to by HEAD), you can use the following command:

**git diff –cached**

To compare two **different commits** in Git, you can use the git diff command with the commit hashes (or references) of the two commits you want to compare.

**git diff <commit1> <commit2>**

**for eg.**

Let’s say you want to compare commit abc1234 with commit def5678. You would run:

**git diff abc1234 def5678**

If you want to compare the latest commit in branch1 with the latest commit in branch2:

**git diff branch1 branch2**

1. **GITHUB**

A diagram of a remote work flow

Description automatically generatedGitHub is a platform for version control and collaboration, allowing developers to host and manage code, track changes, and work together on projects.

**Steps to sync the github repo to local repo**

* **git init**
* **git remote add origin <repository-url>**

**Git push origin master:** git push is used to upload your local repository content to a remote repository. It transfers commits from your local branch to a corresponding branch on a remote repository like github and gitlab.

**Git pull origin master:** git pull is used to fetch and merge changes from the remote repository to your local repository.

**Git Branching:** Git branching is a powerful feature that allows developers to create separate lines of development within a project. Branching enables you to work on different tasks, features, or fixes in isolation from the main codebase.

A **branch** in Git is a pointer to a specific commit in the project’s history. The default branch in Git is called main (or master in older repositories). When you create a new branch, you're essentially creating a new line of development, which starts from a particular commit.

**git branch <branch-name>:**To create a new branch, you use the git branch. This creates a new branch but does not switch to it.

**git checkout -b <branch-name>:**To create and switch to the new branch in one step

You can switch between branches using the **git checkout** or **git switch** command

**git branch:** To see all the branches in your repository.

1. **Merging Branches In GIT**

Merging branch in git is like combining dfiferent versions of a project into one. It allows you to bring changes from one branch into another. This is useful when you’ve been working on a separate branch and now want to integrate those changes into a main branch or another branch.

Steps:

1. Start by creating a new branch where you’ll develop a new feature or fix: **git checkout -b feature-branch.**
2. Work on your changes or new feature in the feature-branch.
3. Save your changes with commits

**git add .**

**git commit -m "Add new feature"**

1. Once your feature is ready, switch back to the main branch.

**git checkout main**

1. Merge the changes from feature-branch into main

**git merge feature-branch**

1. Still the changes is only made in local repository in your master branch, in order to see it in github, we need to push the master branch

**Git push origin master**

1. **Git Conflicts**

A "happy path" merge in Git refers to a straightforward and conflict-free merge process, where the changes from different branches can be combined without any issues.

A **Git conflict** happens when Git can't automatically merge changes from different branches because the same part of a file has been changed in different ways. It's like two people trying to edit the same sentence in a document, but they make different changes, and the software doesn't know which version to keep.

When do conflicts happen

* **Merging branches:** When you combine two branches, and both have made changes to the same lines of code.
* **Rebasing:** When you're updating a branch with changes from another, similar to merging.
* **Cherry-picking:** When you pick a specific change from one branch and apply it to another.

**How do you know theres a conflict?**

Git stops and tells you there's a conflict. It marks the conflicting parts in the file with special lines like **<<<<<<< and >>>>>>>**

1. **Git Rebase**

The first thing to understand about git rebase is that it solves the same problem as git merge.Both of these commands are designed to integrate changes from one branch into another branch. They just do it in very different ways.

**Git rebase** is a command that helps you take the changes you've made in one branch and place them on top of another branch. It’s like moving your work to start after the most recent updates in the project.

* **Keep Your Work Updated:** If other people have made changes in the main branch while you were working on your feature, rebase lets you bring in those changes so your work is up to date.
* **Keep the History Clean:** Rebase makes the project history look neat and linear, without a lot of extra merge commits.

You have two branches in your project: main and feature:

**main: A---B---C---D**

**feature: E---F**

* Commits A, B, C, and D are part of the main branch.
* Commits E and F are part of the feature branch.
* The feature branch was created from main after commit C. Since then, someone else has added commit D to main, and you’ve made two commits (E and F) in feature.

After the rebase is complete, your project history will look like this:

**main: A---B---C---D---E'---F'**

1. **Git rebase false and git rebase ture**

When rebase is set to false, it means that Git will use the default merge strategy when you pull changes from a remote branch.

When rebase is set to true, it means that Git will rebase your current branch on top of the fetched changes from the remote branch instead of merging.

1. **Git Stashing**

Git stashing is like temporarily saving your work without committing it. Imagine you're working on a new feature, but suddenly, you need to switch to another task. You might not be ready to commit your changes yet, so you "stash" them away. This saves your changes in a safe place, allowing you to switch to another branch or pull the latest changes without losing your progress. Later, when you're ready to get back to your work, you can "apply" or "pop" the stash to restore your saved changes.

1. **git stash**: Saves your uncommitted changes and cleans up your working directory.
2. **git stash pop**: Applies the stashed changes back to your working directory and removes them from the stash.
3. **git stash apply**: Applies the stashed changes back, but keeps them in the stash list if you need to use them again later.
4. **Manage multiple stash**
   1. Every time you run git stash, your uncommitted changes are saved in a new stash entry.

**git stash** # Stash your current changes

# Make some more changes

**git stash** # Stash these changes as well

* 1. To view all stashes

**git stash list**

* 1. This command will output a list of all the stashes, something like:

**stash@{0}: WIP on feature-branch: abcdef1 Second stash**

**stash@{1}: WIP on feature-branch: abcdef2 First stash**

* 1. If you want to apply a specific stash from the list, use the index provided by git stash list:

**git stash apply stash@{1}**

* 1. If you no longer need a specific stash and want to remove it without applying, you can drop it:

**git stash drop stash@{1}**

* 1. If you want to clear all stashes:

**git stash clear**

1. **Git Tagging**

Git tagging is a way to mark specific points in your repository’s history as important. Tags are often used to mark release points (e.g., v1.0, v2.0) and other significant commits. They are similar to branches but are meant to be a permanent reference to a commit rather than a moving pointer.

**git tag <tag-name>**

1. **Annoted Tag**

An annotated tag is a special type of tag that includes extra information about the tag, like who created it, when it was created, and a message describing the tag. It’s more detailed than a regular tag and is stored as a complete object in Git.

**git tag -a <tag-name> -m "Tag message"**

**for eg:**

**git tag -a v1.0 -m "Version 1.0 release"**

to see details about tag:

**git show <tag-name>**

To share your tags with others on a remote repository, you need to push them:

**git push origin <tag-name>**

if the tag was already pushed to a remote repository, youll need to force push the updated tag:

**git push - -force origin <tag-name>**

1. **Steps to tag a specific commit in git**
   1. First, you need to identify the hash of the commit you want to tag. You can find it by viewing the commit history:

**git log –oneline**

* 1. Use the git tag command with the -a option to create an annotated tag for a specific commit. Replace <tag-name> with your tag name and <commit-hash> with the hash of the commit you identified:

**git tag -a <tag-name> <commit-hash> -m "Tag message"**

**for eg:**

**git tag -a v1.0 abc1234 -m "Version 1.0 release"**

* 1. To check that the tag has been created and view its details:

**git show <tag-name>**

For eg:

**git show v1.0**

This will display information about the tag, including the commit it points to and the tag message.