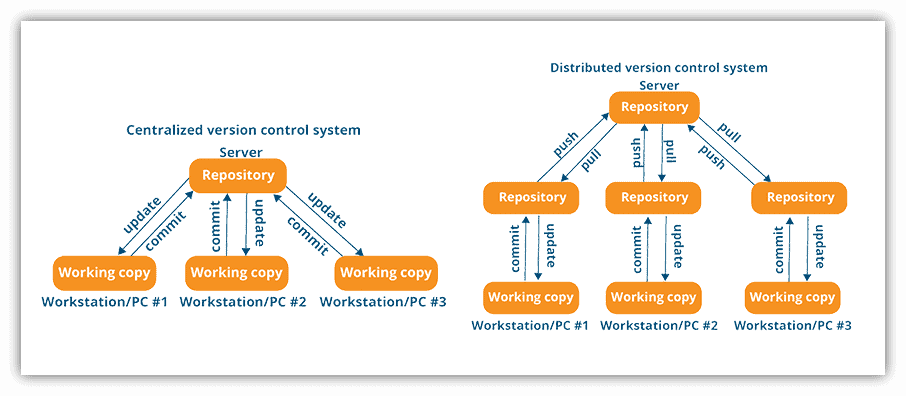
**Version Control System**

A version control system enables multiple software developers to work on a single project together.

* VCS enables this by keeping a track of history of changes.
* VCS enables developers to see every changes. who, what, when.

There are two types of Version Control System in DevOps:

1. **Centralized Version Control System (CVCS)**  
   It uses a central server to store all versions of your code and enables team collaboration. Every developer can copy (check out) a specific version of code on to their computer and then modify it and commit changes to the server.  
   A major drawback of this approach is its single point of failure, that is, the central server.
2. **Distributed Version Control System (DVCS)**  
   In this, every developer has a local copy of the main repository on their computer so if the server goes down then the repository from any client can be copied back to the server and restored.  
   Every checkout is a full backup of the repository.



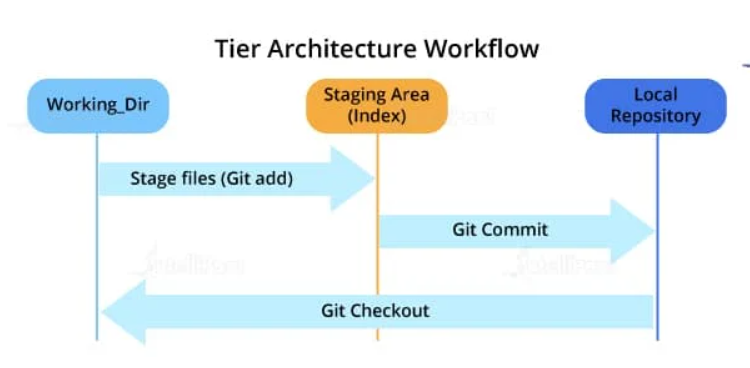
**GIT:**

Git is a powerful and widely-used version control system that allows developers to track changes in their codebase, collaborate seamlessly, and maintain a robust history of their projects. It was created by **Linus Torvalds** in 2005, Git has become widely accepted for version control in the software development industry.

**Why We Use Git:**

* Version Control
* Collaboration
* Backup and Restore

**Git Architecture**



Most of the version control systems have a two-tier architecture (Central version control system that is working and local repository). However, Git has a layer more, making it a three-tier architecture. But, why are there three layers of Git? Let’s find out.

**The three layers are:**

* **Working directory**: This is created when a Git project is initialized onto your local machine and allows you to edit the source code copied.
* **Staging area**: Post the edits, the code is staged in the staging area by applying the command, **git add**. This displays a preview for the next stage. In case further modifications are made in the working directory, the snapshots for these two layers will be different. However, these can be synced by using the same ‘git add’ command.

**Git add .** (. Means path of the file)

* **Local repository**: If no further edits are required to be done, then you can go ahead and apply the **git commit** command. This replicates the latest snapshots in all three stages, making them in sync with each other.

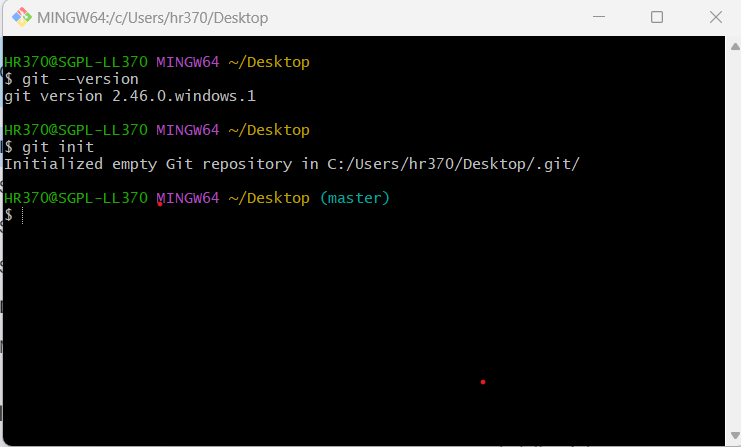
**Git commit -**m “message name”

**CLI – COMMOND LINE INTERFACE:**

**CLI** stands for **Command Line Interface**. It is a way to interact with your computer or software by typing commands into a text-based interface, rather than using graphical user interfaces (GUIs) with buttons and menus.

**Simple Explanation:**

* **Command Line**: Instead of clicking on icons or menus, you type commands into a terminal or console window.
* **Interface**: This is the method you use to communicate with your software or operating system.

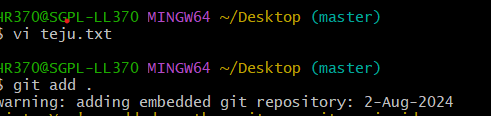


**Basic Git Commands:**

* **git init**: This Git command converts a directory into an empty repository. This is the initial step you need to take to build a repository. Once you run git init, you will be able to add and commit files and directories.



* **git add:** This command allows you to add files in the Git staging area. The file must be added to the index of Git before being available to commit to any particular repository. You can use this command to add directories, files, etc.



* **git commit**: The commit command in Git allows you to track the changes in the files in a local repository. Each commit has its own unique ID for reference.



* **git status:** The git status command returns the present state of a repository, like if the file is in the staging area but has not been committed.



* **git config:** There are numerous configurations and settings possible in Git, and this command allows you to assign these settings. User.name and user. email are the two significant settings that set the name and email address of a user.
  1. **Set User name:**

**git config --global user.name "Your Name"**

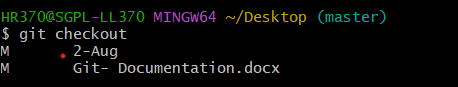
* 1. **Set User Email:**

**git config --global user.email "your.email@example.com"**

* **git branch**: This command determines the branch of the local repository and allows you to add or delete a branch.



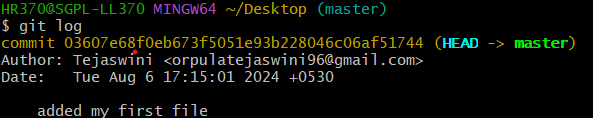
* **git checkout:** You can use this command to switch to another branch.



* **git merge:** The merge command allows you to integrate two or more branches together. It combines the changes made in the branches.

**git merge feature-branch**

* **git log:** To check the commit details



**Git Commands When Working with Remote Repositories:**

1. **git remote:** This Git command allows you to connect a remote repository to a local repository.
2. **git clone:** You can use the clone command to create a local copy of an already existing remote repository. This allows you to copy and download the required repository to the system. It is similar to the init command while working with remote repositories as it allows you to build a local directory, consisting of all the necessary files and history of the repository.
   1. **Clone a Repository from GitHub**:

**git clone https://github.com/username/repository.git**

1. **git pull:** The pull command is used to run the latest version of any repository. This pulls all the changes made from the remote to the local repository.

\* **git fetch**: Downloads new data from a remote repository but does not automatically merge it into your working branch.

\* **git merge**: Integrates the fetched changes into your current branch.

**git fetch + git merge = git pull**

**Example:**

1. **Pull Changes from the Default Remote (origin) and Branch:**

**git pull**

1. **Pull Changes from a Specific Remote:**

**git pull origin**

**3, Pull Changes from a Specific Branch:**

**git pull origin main**

1. **git push:** This command sends local commits to the respective remote repository. It needs two parameters, i.e., the remote repository and the specific branch where it needs to be pushed.

**Essentials of GIT in industry:**

In the industry, Git is widely adopted for version control due to its powerful features and flexibility. Here are some of the essentials of Git that are particularly valuable in a professional or production environment:

1. **Branching and Merging**

* Branching: Git allows you to create branches, enabling parallel development. This is crucial for features, bug fixes, and experimentation without affecting the main codebase.
  + **Example**: You might create a branch for a new feature (git checkout -b feature/new-feature) or (git branch <branch name>).
* Merging: Once work on a branch is completed, you merge it back into the main branch (often main or master), integrating the changes.
  + Example: git merge feature/new-feature merges the feature/new-feature branch into the current branch.

2. **Commit History and Version Control**

* Commits: Git records changes in commits, each with a unique ID, author information, and a message describing the change.
  + **Example**: git commit -m "Fix bug in authentication module"
* History: You can view the history of commits with git log, which helps track changes and understand the evolution of the codebase.

3. **Staging Area (Index)**

* Staging: The staging area allows you to prepare changes before committing. You can add specific files or portions of files to the staging area.
  + **Example**: git add file.txt stages file.txt for the next commit.

4. **Remote Repositories**

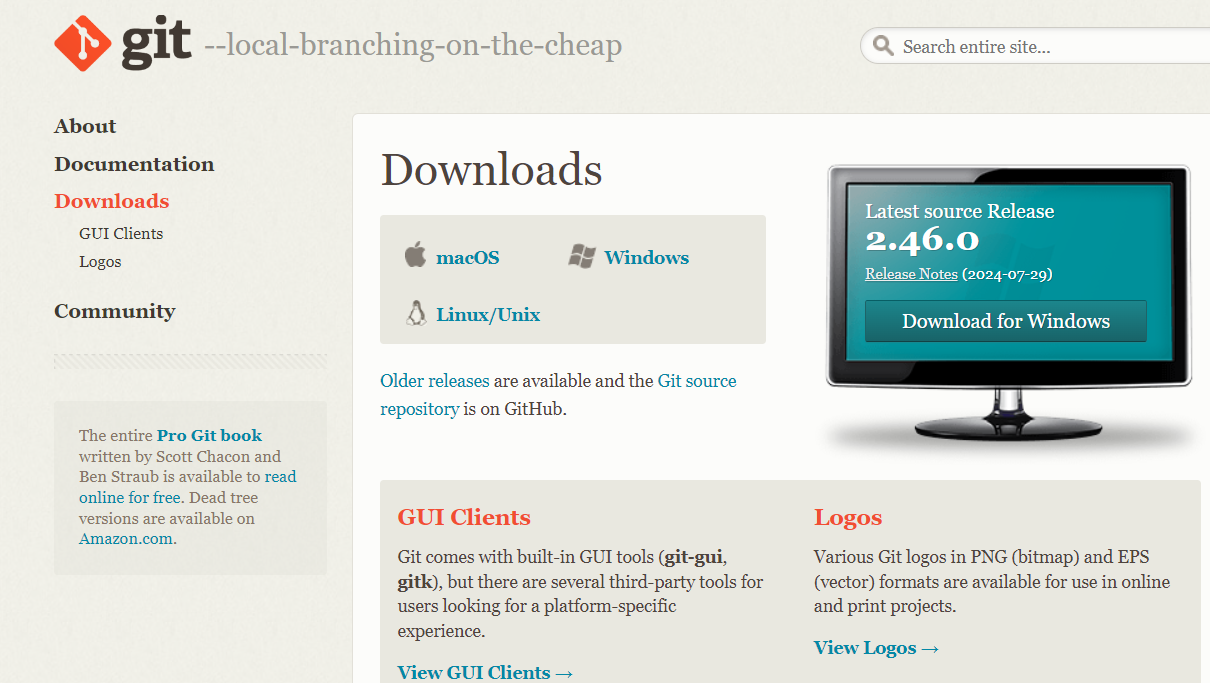
* Remote Repositories: Git supports working with remote repositories hosted on platforms like GitHub, GitLab, or Bitbucket. This is essential for collaboration and backup.
  + **Example**: git push origin main uploads local changes to the remote repository.
* Cloning and Pulling: You can clone a remote repository to your local machine (git clone <url>) and fetch updates from the remote repository (git pull).

1. **Tagging**

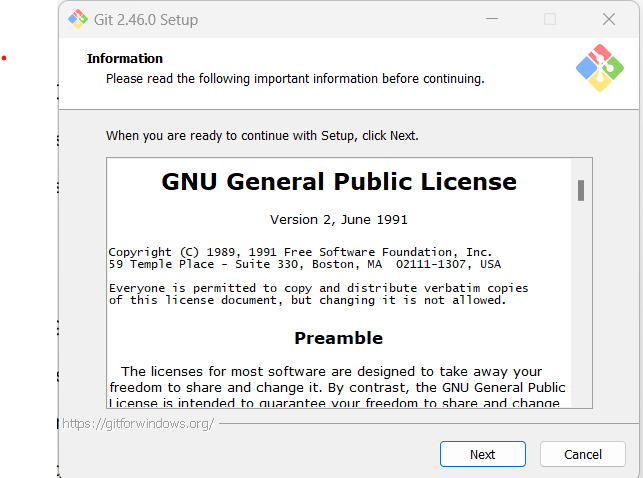
* **Tags**: Git allows you to tag specific commits with meaningful names, often used to mark release versions.
  + **Example**: git tag v1.0 creates a tag named v1.0 for the current commit.

**How to setup GIT**

**GIT Installation:**

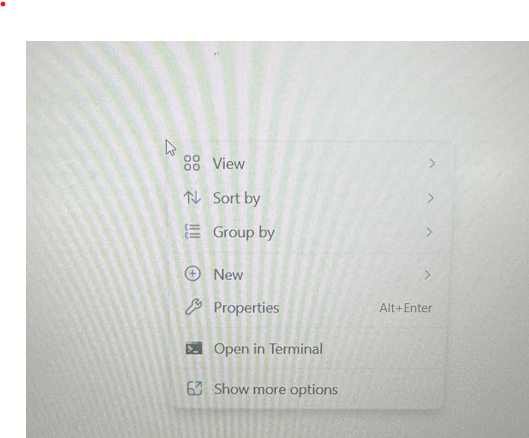
****

Once you click on download, the below screen will appear so click on “Next” to setup

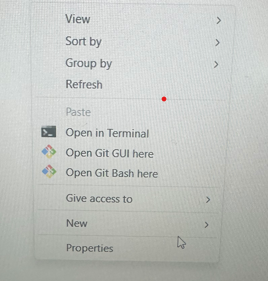
****

Once the above step is completed, Check whether the git is installed or not:

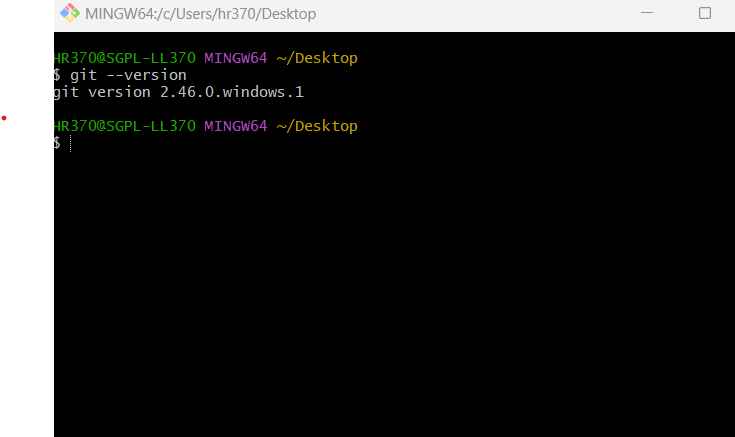
Click on Show more options as shown below:



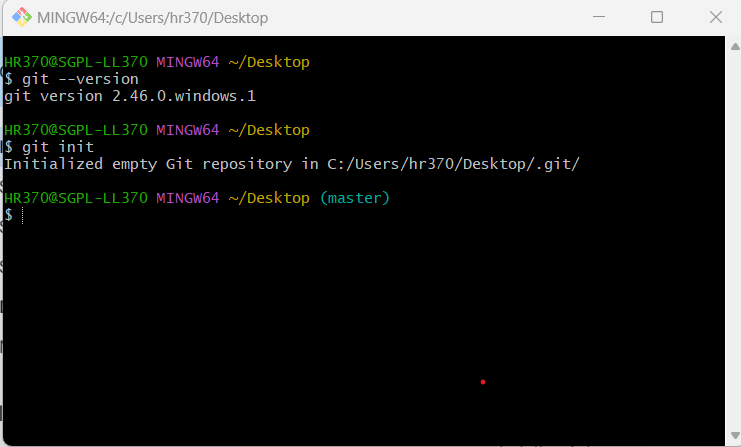
If you were able to see the below Open Git GUI here and Open Git Bash here, Then git installed successfully:



Check the git version in terminal:



Initially, a local repository



**Working with various commands in GIT:**

1. **git init**
   * Initializes a new Git repository in the current directory.
2. **git clone <repository\_url>**
   * Creates a local copy of a remote repository.
3. **git add <file>**
   * Stages a file for commit.
4. **git commit -m "message"**
   * Commits the staged changes with a message.
5. **git status**
   * Displays the status of your working directory and staging area.
6. **git log**
   * Shows the commit history.

**Branching and Merging**

1. **git branch**
   * Lists all branches in the repository.
2. **git branch <branch name>**
   * Creates a new branch.
3. **git checkout <branch name>**
   * Switches to a different branch.
4. **git checkout -b <branch name>**
   * Creates and switches to a new branch.
5. **git merge <branch name>**
   * Merges the specified branch into the current branch.
6. **git rebase <branch name>**
   * Reapplies commits from the current branch on top of the specified branch.

**Remote** **Repositories**

1. **git remote -v**
   * Lists remote repositories associated with your local repository.
2. **git remote add <name> <repository URL>**
   * Adds a new remote repository.
3. **git fetch <remote>**
   * Fetches updates from a remote repository but does not merge them.
4. **git pull**
   * Fetches and merges changes from the remote repository into the current branch.
5. **git push <remote> <branch>**
   * Pushes changes to a remote repository.

**Undoing Changes**

1. **git reset <file>**
   * Unstages a file from the staging area.
2. **git reset --hard**
   * Resets the working directory and staging area to the last commit, discarding all changes.
3. **git revert <commit>**
   * Creates a new commit that undoes the changes from a specific commit.
4. **git stash**
   * Temporarily saves changes in a "stash" to be reapplied later.
5. **git stash pop**
   * Applies the most recent stash and removes it from the stash list.

**Viewing and Comparing Changes**

1. **git diff**
   * Shows differences between the working directory and the staging area.
2. **git diff --staged**
   * Shows differences between the staging area and the last commit.
3. **git diff <branch1> <branch2>**
   * Shows differences between two branches.

**Configuration and Information**

1. **git config --global user.name "Your Name"**
   * Sets the global username for commits.
2. **git config --global user.email "your.email@example.com"**
   * Sets the global email address for commits.
3. **git config --list**
   * Lists all Git configuration settings.

**Advanced Commands**

1. **git cherry-pick <commit>**
   * Applies the changes from a specific commit to the current branch.
2. **git tag <tag\_name>**
   * Creates a new tag for a commit.
3. **git tag -d <tag\_name>**
   * Deletes a tag.
4. **git reflog**
   * Shows a log of all the recent actions (useful for recovering lost commits).
5. **git blame <file>**
   * Shows who last modified each line of a file.

**Recording Changes to the Repository**

·       A.  How to check the Status of Your Files

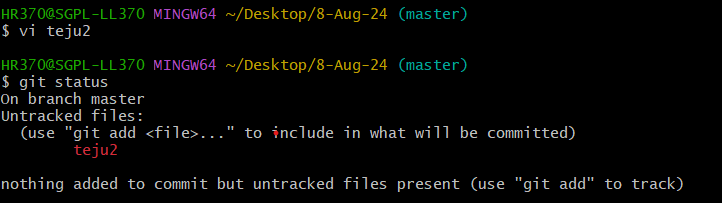
Primarily use the git status command. This command gives you a comprehensive overview of the current state of your working directory and staging area.

**git status**

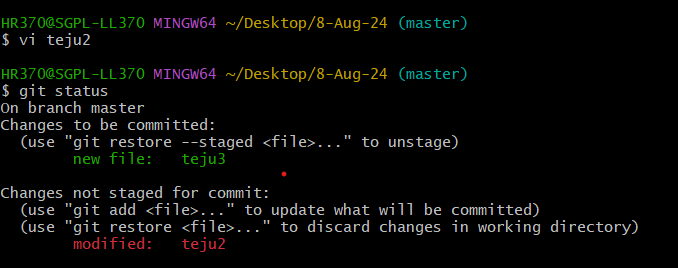
The git status output is divided into several sections:

1. **Untracked files: These are files that are not currently tracked by Git. They are new files or files that have been added to your working directory but not yet staged for commit.**

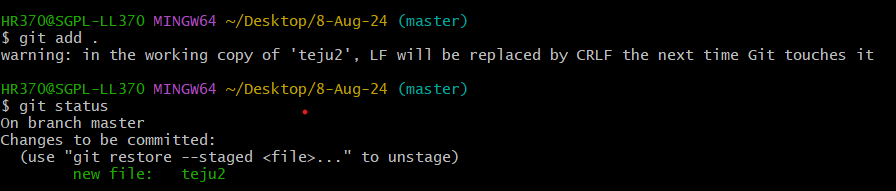
**Make file**



1. **Changes not staged for commit--You need to add these files to the staging area before committing.**



1. **Changes to be committed-- These are files that have been staged and are ready to be `committed. They were modified and then added to the staging area using git add.**



**How to Stage our modified files in git**

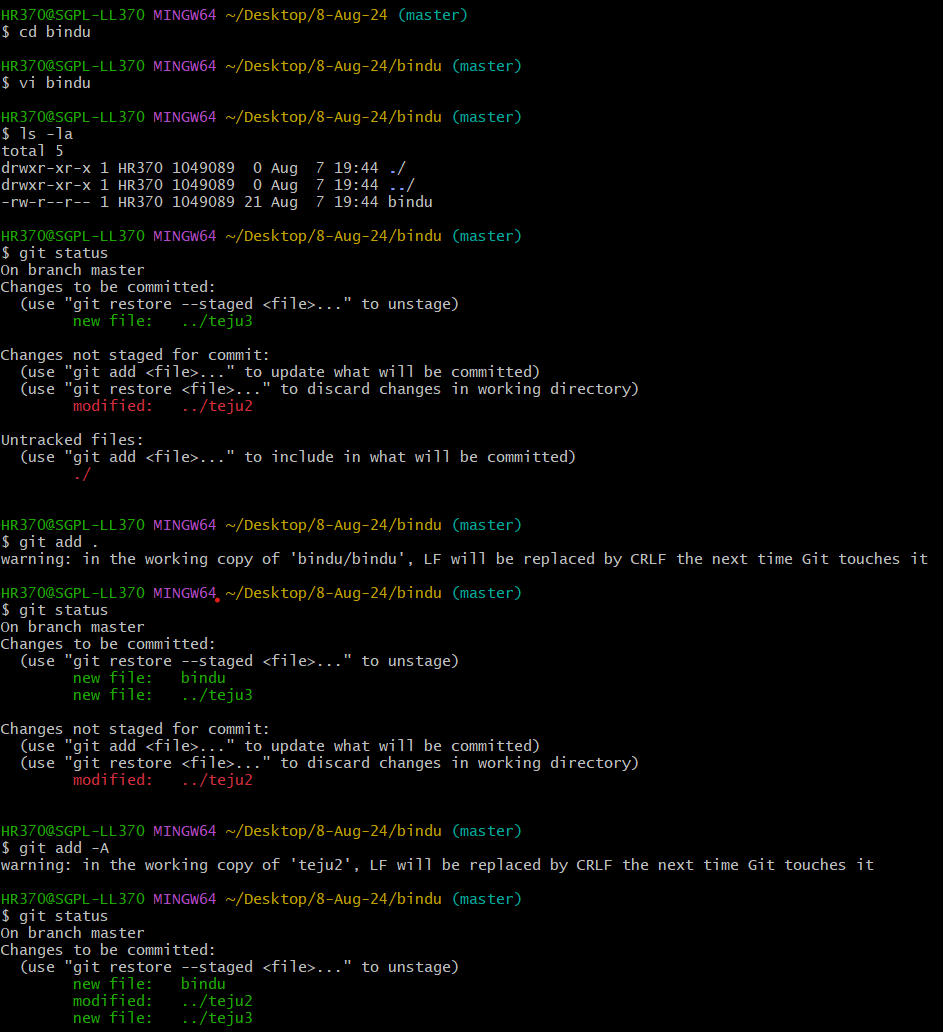
* To stage all modified files at once, use:

**git add .**

This command stages all changes (new, modified, and deleted files) in the current directory and its subdirectories.

* If you want to stage all changes across the entire repository:

**git add -A**

This stages all changes in the working directory, including modifications, new files, and deletions.

**How to ignore files in git :**

To ignore files in Git, you use .gitignore file. This file tells Git which files or directories to ignore, preventing them from being tracked or included in commits.

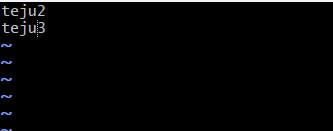
* + 1. Create a .gitignore file

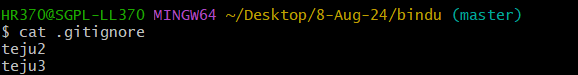
**touch .gitignore**

****

* + 1. **Open the .gitignore file and add the file name that should be ignored. By adding them into this .gitignore file**

****

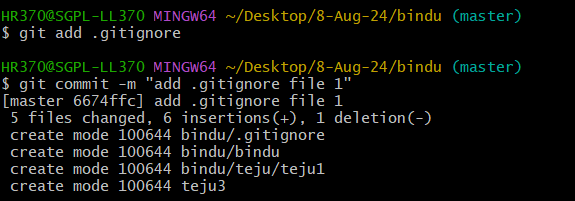
****

****

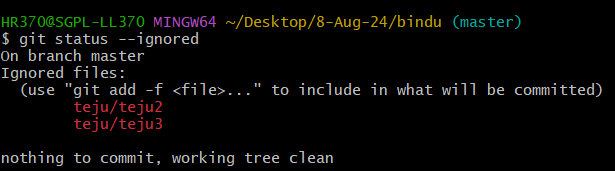
* + 1. **Commit the .gitignore File**

**git add .gitignore**

**git commit -m "Add .gitignore file"**

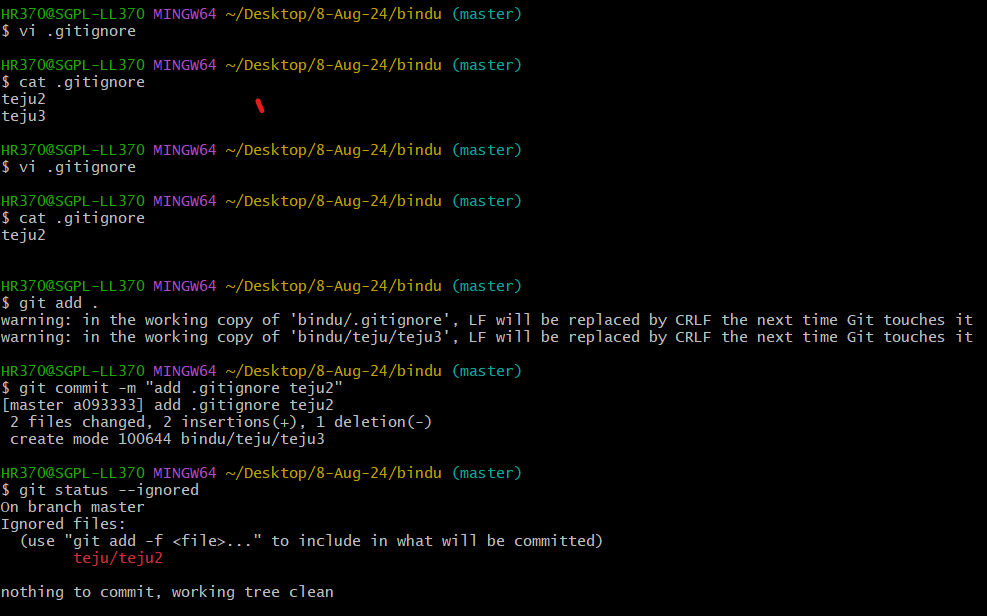
****

### **Verify Ignored File:**To see which files are being ignored, you can use:



**If you want to undo the ignore files, Follow the below steps:**

* + 1. Vi .gitignore – Remove the file name
    2. git add .gitignore
    3. git commit -m “add .gitignore teju2”
    4. git status --ignored

****

**Viewing Your Unstaged and Staged Changes in git**

1. **Viewing Unstaged Changes:**

**git diff**

* Shows the changes between the working directory and the staging area.
* This command displays what has been modified but not yet staged for commit.

**How to commit Your Changes:**

**1. Stage Your Changes**: Before committing, you need to stage the changes you want to include in the commit. Use the git add command to do this.

**git add <file1> <file2> ...**

**To stage all changes in the working directory:**

**git add .**

**2.Commit the Changes**: Once your changes are staged, you can commit them using the git commit command. Include a meaningful commit message that describes the changes.

**git commit -m "Your commit message here"**

**3.Push the Changes** (Optional): If you're working with a remote repository and want to upload your commits, use the git push command.

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

**Skipping the Staging Area and commit:**

**Commit All Changes Directly**

To skip the staging area and commit all changes to tracked files directly, use:

bash

**git commit -a -m "Your commit message here"**

**Removing Files from GIT:**

**1.Removing a File from the Working Directory and Git Repository:**

**git rm <file>**

1. **Remove a File from the Repository but Keep It Locally**

**git rm --cached <file>**

**Viewing the Commit History:**

**1. Basic Commit History:**

**Git log**

1. **Compact and One-Line View**

**git log –oneline**

1. **Filtering Commits**

**git log --author="Author Name"**

**git log --since="2024-08-01" --until="2024-08-31"**

1. **Viewing a Specific Commit**

**git show <commit-ID>**

1. **Viewing Commit History with Differences**

**git log -p**

**Web-based Git services offer visual commit history through their interfaces.**

**GitHub**

* **Features:**
  + **Commits Tab: Shows a graphical view of commit history and branches.**
  + **Pull Requests: Manage and view pull requests with associated commits.**
* **How to Use:**
  + **Open Your Repository: On GitHub.**
  + **View Commits: Go to the Commits tab to see a list of commits with a graphical history.**
  + **This git tool provide a range of options for visualizing and managing Git commit history, making it easier to understand and work with your repository's history.**

**Undoing Things:**

**1. Changing Your Last/Latest Commit:**

Changing the Last/Latest Commit Message:

**git commit –amend**

**2. Modifying the Last Commit’s Content**

Make the changes to the files you want to adjust.

Stage the changes

**git add <file>**

Amend the commit:

**git commit –amend**

**3. Undoing the Last Commit**

If you want to undo the last commit but keep your changes, you can use:

**git reset --soft HEAD~1**

* + **Force Pushing After Amending:**
* If you have already pushed the commit to a remote repository and you amend it, you will need to force push to update the remote branch:

**git push –force**

**Unstaging a Staged File**

If you have a file that you've staged for a commit but now want to unstage, you can use the git reset command to remove it from the staging area while keeping the changes in your working directory. Here’s how you can do it:

**Unstaging a Specific File:**

To unstage a specific file that you've staged, use:

**git reset <file>**

This command will unstage the file, meaning it will be removed from the staging area, but any changes you made to the file will still be present in your working directory.

**Unstaging All Staged Files:**

If you want to unstage all files that have been staged, you can use:

**git reset**

This will unstage all files that have been added to the staging area, but again, your changes will remain in your working directory.

Check the Status

To see which files are currently staged and which are not, you can use:

**git status**

This will show you a list of staged changes, changes in your working directory that are not yet staged, and untracked files.

**Unmodifying a Modified File:**

If you’ve modified a file in your working directory and you want to revert it to the state of the last commit (i.e., discard your changes), you can use Git commands to achieve this. Here’s how to do it:

**1. Discard Changes in a Single File**

To revert a specific file to its last committed state, use:

git checkout -- <file>

Or in newer versions of Git:

git restore <file>

**2. Discard Changes in All Modified Files:**

If you want to discard changes in all modified files (but keep untracked files intact), you can use:

**git checkout -- .**

Or in newer versions of Git:

**git restore .**

**3.Discard Changes in Staged Files:**

If you have staged changes (added to the staging area) and you want to unstage them and revert the files to their last committed state, follow these steps:

1. **Unstage the files:**

**git reset <file>**

Or to unstage all staged files:

**git reset**

1. **Discard changes:**

Use the appropriate command to revert the files:

**git checkout -- <file>**

or

**git restore <file>**

1. **Discard Changes in Staged Files**

If you have staged changes (added to the staging area) and you want to unstage them and revert the files to their last committed state, follow these steps:

1. **Unstage the files:**

**git reset <file>**

**Or to unstage all staged files:**

**git reset**

1. **Discard changes:**

**Use the appropriate command to revert the files:**

**git checkout -- <file>**

**or**

**git restore <file>**

**Working with Remotes**

To manage and view the remote repositories associated with your local Git repository, you can use several commands. Here’s how to show and work with remotes:

**1. Listing Remote Repositories**

To view the list of remote repositories and their URLs, use:

**git remote -v**

This command will display the remote names and their corresponding URLs, showing both fetch and push URLs.

**2. Detailed Remote Information**

For more detailed information about a specific remote, use:

**git remote show <remote-name>**

Replace <remote-name> with the name of the remote (usually origin if you haven’t added additional remotes). This command provides detailed information about the remote, including:

* The remote URL
* The tracking branches
* The local branches configured to push and fetch from this remote

**3. Display Remote Configuration**

To see the configuration for all remotes, including URLs and other settings, use:

**git remote show**

This will list the names of all configured remotes.

**4. List Branches and Tracking Information**

To see which branches are tracking remote branches, use:

**git branch -vv**

This will display local branches along with information about their upstream (remote) branches and the commit messages of the latest commits.

**Adding Remote Repositories:**

Adding remote repositories in Git allows you to connect your local repository to a remote server, where you can push your changes and pull updates. Here’s how to add and manage remote repositories:

**1. Adding a New Remote Repository**

To add a new remote repository, use the git remote add command. The basic syntax is:

**git remote add <remote-name> <remote-url>**

* **<remote-name>: A name for the remote repository (e.g., origin or upstream).**
* **<remote-url>: The URL of the remote repository.**

This command adds a remote named origin that points to the specified URL.

**2. Verifying the Remote Repository**

To verify that the remote was added correctly, you can list the remotes:

**git remote -v**

This will display the remote names and their corresponding URLs.

**3. Changing the URL of an Existing Remote**

If you need to update the URL for an existing remote, use:

**git remote set-url <remote-name> <new-remote-url>**

**4. Removing a Remote Repository**

To remove a remote repository, use:

**git remote remove <remote-name>**

**5. Renaming a Remote Repository**

If you need to rename an existing remote, use:

**git remote rename <old-remote-name> <new-remote-name>**

**This renames the remote from origin to upstream.**

**Fetching and Pulling from Your Remotes**

Fetching and pulling from remote repositories in Git are crucial for synchronizing your local repository with changes made in remote repositories. Here’s how you can effectively use these commands:

**Fetching from a Remote**

The git fetch command downloads commits, files, and refs from a remote repository to your local repository without merging those changes into your working directory. This is useful to see what changes are available in the remote without integrating them into your current branch.

**git fetch <remote-name>**

This command fetches updates from the remote repository named origin. It updates your local copy of the remote branches but does not modify your working directory or current branch.

Checking the Fetched Changes

To see what has been fetched, you can check the log or status:

**git log <branch-name>..<remote-name>/<branch-name>**

**Pulling from a Remote**

The git pull command combines git fetch and git merge into a single step. It fetches changes from the remote repository and then merges them into your current branch.

**git pull <remote-name> <branch-name>**

**Handling Conflicts**

If there are conflicts between your local changes and the fetched changes, Git will notify you and you’ll need to resolve these conflicts manually. After resolving conflicts, you can complete the merge with:

**git add <resolved-file>**

**git commit**

**Pushing to Your Remotes**

Pushing changes to remote repositories in Git is how you upload your local commits to a remote server, making them available to other collaborators. Here’s a detailed guide onhow to push changes to remotes and manage this process effectively:

Basic Push Command

To push your changes to a remote repository, use:

**git push <remote-name> <branch-name>**

* **<**remote-name>: The name of the remote repository (e.g., origin).
* <branch-name>: The branch you want to push to (e.g., main or feature-branch).

### **Pushing to a Remote and Tracking a Branch**

When you create a new local branch and want to push it to a remote for the first time, you can use:

**git push -u <remote-name> <branch-name>**

The -u (or --set-upstream) flag sets the upstream tracking relationship, meaning that future pushes and pulls will default to this remote branch.

**Force Pushing**

If you need to overwrite changes on the remote branch (e.g., after rewriting history with git rebase), you can use:

git push --force <remote-name> <branch-name>

**Caution:** Force pushing can overwrite changes in the remote repository and affect other collaborators. Use this with care and communicate with your team.

**Pushing Tags**

To push tags to a remote repository, you can use:

git push <remote-name> <tag-name>

**Pushing All Branches**

To push all local branches to the remote:

git push --all <remote-name>

**Inspecting a Remote**

Inspecting a remote in Git involves checking details about the remote repositories associated with your local repository. This helps you understand the remote's configuration, the branches it has, and how your local branches interact with it. Here are the key commands for inspecting a remote:

**1. List All Remotes**

To list the names of all configured remotes:

git remote

To see the URLs associated with each remote (both fetch and push URLs):

git remote -v

**2. Detailed Remote Information**

To get detailed information about a specific remote:

git remote show <remote-name>

Replace <remote-name> with the name of the remote (e.g., origin).

git remote show origin

This command provides detailed information, including:

* **Fetch URL:** The URL used to fetch changes from the remote.
* **Push URL:** The URL used to push changes to the remote.
* **HEAD branch:** The default branch of the remote repository.
* **Remote branches:** A list of branches in the remote repository.
* **Local ref configured for 'git pull':** The local branch that is configured to track the remote branch and perform merges.
* **Local ref configured for 'git push':** The local branch that is configured to push changes to the remote branch.

**3. View Remote URL Configuration**

To see the URL configured for a specific remote:

git remote get-url <remote-name>

git remote get-url origin

This will display the URL of the remote named origin.

**4. View All Remote URLs and Settings**

To view all remote URLs and configuration details:

git remote -v

This provides a simple list of remote names with their associated URLs for both fetch and push operations.

**5. List Remote Branches**

To list all branches available on a specific remote:

git branch -r

This lists all remote-tracking branches. You can see branches like origin/main or origin/feature-branch.

To list branches from a specific remote:

git branch -r | grep <remote-name>/

**Removing and Renaming Remotes**

In Git, you might need to remove or rename remote repositories depending on changes in your workflow or repository configuration. Here’s how to handle these tasks:

**Removing a Remote**

To remove a remote repository from your local Git configuration, use:

git remote remove <remote-name>

Alternatively, you can use:

git remote rm <remote-name>

Both commands accomplish the same task.

**Renaming a Remote**

To rename an existing remote, use:

git remote rename <old-remote-name> <new-remote-name>

**Branching and Merging in Git**

In Git, a branch is a fundamental concept that allows you to diverge from the main line of development and continue to work in isolation. Here’s a detailed explanation of what a branch is and why it’s useful:

**What is a Branch?**

A branch in Git is essentially a pointer to a specific commit in your repository. Branches allow you to work on different features, bug fixes, or experiments concurrently without affecting the main codebase.

**Key Characteristics of a Branch**

1. **Pointer to a Commit:**
   * Each branch is a reference to a specific commit. The branch pointer moves forward as new commits are added.
   * For example, when you create a new branch, it points to the same commit as the branch you were on when you created it.
2. **Isolated Development:**
   * Changes made in one branch do not affect other branches. This isolation makes it easy to develop features or fix bugs without interfering with the main codebase.
3. **Flexible and Lightweight:**
   * Branches are lightweight and can be created, deleted, or switched with minimal overhead. Creating and switching between branches is fast and efficient.
4. **Used for Various Purposes:**
   * Branches can be used for various purposes, such as developing new features, fixing bugs, experimenting with new ideas, or preparing releases.

**Common Branch Types**

1. **Main Branch (or Master):**
   * The default branch created when you initialize a Git repository. Often used as the stable branch or the production-ready branch.
2. **Feature Branches:**
   * Branches created to develop new features. Typically named to reflect the feature being developed (e.g., feature/login-form).
3. **Bugfix Branches:**
   * Branches used to fix bugs or issues. They are often named to indicate the nature of the bug being fixed (e.g., bugfix/login-error).
4. **Release Branches:**
   * Branches used to prepare for a new release. They help in stabilizing the code before the release (e.g., release/v1.0).
5. **Hotfix Branches:**
   * Branches created to quickly address critical issues in the production code (e.g., hotfix/security-patch).

**Basic Branch Commands**

* **List All Branches:**

git branch

Shows all local branches. To list remote branches as well:

git branch -a

* **Create a New Branch:**

git branch <branch-name>

Creates a new branch. To create and switch to it immediately:

git checkout -b <branch-name>

* **Switch to a Branch:**

git checkout <branch-name>

Or using the newer command:

git switch <branch-name>

* **Delete a Branch:**

git branch -d <branch-name>

Deletes a branch that has been merged. To force delete a branch (even if it hasn’t been merged):

git branch -D <branch-name>

**Basic in Branching and Merging**

Branching and merging are core features of Git that enable efficient and flexible management of changes in your codebase. Here’s a basic guide to understanding and using branching and merging in Git:

**1. Branching**

**Creating a Branch**

To create a new branch:

git branch <branch-name>

This command creates a new branch but does not switch to it. For example:

git branch feature/new-feature

**Switching Branches**

To switch to a different branch:

git checkout <branch-name>

Or using the newer command:

git switch <branch-name>

git checkout feature/new-feature

**Creating and Switching to a Branch**

You can create a new branch and switch to it in a single command:

git checkout -b <branch-name>

or

git switch -c <branch-name>

git checkout -b feature/new-feature

**Listing Branches**

To list all local branches:

git branch

To list all branches, including remote branches:

git branch -a

**Deleting a Branch**

To delete a branch that has been merged:

git branch -d <branch-name>

To force delete a branch (even if it hasn't been merged):

git branch -D <branch-name>

**2. Merging**

**Merging a Branch**

To merge changes from one branch into another, first, switch to the branch you want to merge into (usually main or master):

git checkout <target-branch>

Then, merge the branch with the changes:

git merge <source-branch>

**Handling Merge Conflicts**

If there are conflicts between the branches, Git will notify you during the merge. You need to resolve these conflicts manually:

1. Open the conflicting files and make necessary changes.
2. After resolving conflicts, stage the resolved files:

git add <resolved-file>

1. Complete the merge by committing the changes:

git commit

**Abort a Merge**

If you want to abort a merge and return to the state before the merge started:

git merge --abort

**3. Basic Workflow**

1. **Start New Feature:**

git checkout -b feature/my-feature

1. **Make Changes and Commit:**

git add .

git commit -m "Add my feature"

1. **Switch Back to Main Branch:**

git checkout main

1. **Merge Feature Branch into Main:**

git merge feature/my-feature

1. **Delete the Feature Branch (if no longer needed):**

git branch -d feature/my-feature

 Branch Management in GIT

Branch management in Git involves creating, organizing, and maintaining branches to ensure an efficient workflow and clear organization of code changes. Here’s a guide on various branch management tasks and best practices:

**1. Creating and Switching Branches**

**Create a New Branch**

To create a new branch without switching to it:

git branch <branch-name>

**Create and Switch to a New Branch**

To create a new branch and immediately switch to it:

git checkout -b <branch-name>

or

git switch -c <branch-name>

**2. Listing Branches**

**List Local Branches**

To view all local branches:

git branch

**List All Branches**

To view all branches, including remote-tracking branches:

git branch -a

**List Remote Branches**

To view branches on a specific remote:

git branch -r

**3. Renaming Branches**

**Rename the Current Branch**

To rename the branch you are currently on:

git branch -m <new-branch-name>

**Rename a Different Branch**

To rename a branch that you are not currently on:

git branch -m <old-branch-name> <new-branch-name>

**4. Deleting Branches**

**Delete a Local Branch**

To delete a local branch that has been merged:

git branch -d <branch-name>

To force delete a branch (even if it hasn’t been merged):

git branch -D <branch-name>

**Delete a Remote Branch**

To delete a branch from a remote repository:

git push <remote-name> --delete <branch-name>

**5. Merging Branches**

**Merge a Branch into Another**

Switch to the branch you want to merge into, and then:

git merge <branch-to-merge>

**Resolve Merge Conflicts**

If conflicts arise:

1. Open the conflicting files and resolve the conflicts.
2. Stage the resolved files:

git add <resolved-file>

1. Commit the merge:

git commit

**6. Rebasing Branches**

**Rebase a Branch onto Another**

Rebasing allows you to move or combine a sequence of commits to a new base commit:

git rebase <branch-to-rebase-onto>

**Handling Rebase Conflicts**

1. Resolve conflicts in the same way as during a merge.
2. Continue the rebase after resolving conflicts:

git rebase --continue

1. If needed, abort the rebase:

git rebase --abort

**7. Tracking Remote Branches**

**Create a Local Branch Tracking a Remote Branch**

To create a local branch that tracks a remote branch:

git checkout -b <branch-name> <remote-name>/<remote-branch-name>

Or:

git switch -t <remote-name>/<remote-branch-name>

**Update Local Tracking Branch**

To fetch changes from the remote branch and update your local branch:

git pull

**Branching Workflows and its usage**

Branching workflows in Git are strategies for managing development efforts and integrating code changes efficiently. Different workflows suit different projects and team sizes. Here’s an overview of common branching workflows and their usage:

**1. Git Flow Workflow**

**Overview:** Git Flow is a branching model that defines a set of rules and procedures for managing branches in a repository. It’s suited for larger projects with scheduled releases and multiple environments.

**Key Branches:**

* **main (or master):** Contains the production-ready code. Only stable, release-ready code is merged into this branch.
* **develop:** Integrates features and fixes. This is the main branch for ongoing development.
* **Feature Branches:** Created from develop for new features or improvements. Merged back into develop when complete.
* **Release Branches:** Created from develop when preparing for a release. Used for final adjustments and bug fixes before merging into main and develop.
* **Hotfix Branches:** Created from main to quickly address critical issues in production. Merged into both main and develop.

**Usage:**

1. **Feature Development:** Create feature branches from develop:

git checkout -b feature/feature-name develop

1. **Release Preparation:** Create a release branch from develop:

git checkout -b release/v1.0 develop

1. **Hotfixes:** Create hotfix branches from main:

git checkout -b hotfix/issue-description main

**Advantages:**

* Well-structured with clear roles for each branch.
* Supports parallel development and multiple releases.

**Disadvantages:**

* Can be complex for small teams or simple projects.
* May involve many branches to manage.

**2. GitHub Flow**

**Overview:** GitHub Flow is a simpler workflow that is well-suited for continuous delivery and agile development. It’s often used for projects that deploy frequently.

**Key Branches:**

* **main (or master):** The stable branch that is always deployable.

**Usage:**

1. **Create a Branch:** Create a new branch for your work:

git checkout -b branch-name

1. **Work and Commit:** Make changes and commit them to your branch.
2. **Push Branch:** Push your branch to the remote repository:

git push origin branch-name

1. **Create a Pull Request:** Open a pull request to merge changes into main.
2. **Review and Merge:** Review the pull request, resolve conflicts if necessary, and merge it.

**Advantages:**

* Simpler and more streamlined.
* Encourages frequent, small changes and continuous deployment.

**Disadvantages:**

* Less structured than Git Flow.
* May not be suitable for complex release management.

**3. GitLab Flow**

**Overview:** GitLab Flow integrates ideas from Git Flow and GitHub Flow, adding features for environments and issue tracking. It’s designed to support various deployment strategies.

**Key Branches:**

* **main (or master):** Contains the stable, production-ready code.
* **Environment Branches:** Branches for specific deployment environments (e.g., staging, production).

**Usage:**

1. **Feature Branches:** Develop features on separate branches.
2. **Environment Branches:** Create branches for staging or production environments.
3. **Merge Requests:** Use merge requests to integrate changes and deploy to specific environments.

**Advantages:**

* Flexible with different deployment strategies.
* Supports environments and issue tracking.

**Disadvantages:**

* Complexity can increase with more environments.
* Requires integration with issue tracking tools.

**Remote Branches – create and delete**

Managing remote branches in Git involves creating and deleting branches on a remote repository. Here's how you can handle these tasks:

**Creating Remote Branches**

To create a remote branch, you generally follow these steps:

1. **Create a Local Branch:**

First, create a local branch that you want to push to the remote repository:

git checkout -b <branch-name>

1. **Push the Local Branch to Remote:**

Push the local branch to the remote repository:

git push origin <branch-name>

This command creates a branch named feature/new-feature on the remote repository (origin in this case) and pushes your local commits to it.

1. **Set Upstream Tracking (Optional):**

If you want the local branch to track the remote branch (so you can use git pull and git push without specifying the remote and branch), you can set the upstream branch:

git push --set-upstream origin <branch-name>

git push --set-upstream origin feature/new-feature

**Deleting Remote Branches**

To delete a remote branch, follow these steps:

1. **Delete the Branch from Remote:**

Use the git push command with the --delete option to remove a branch from the remote repository:

git push <remote-name> --delete <branch-name>

This command removes the branch named feature/old-feature from the origin remote.

1. **Delete the Local Tracking Branch (Optional):**

If you have a local branch that tracks the deleted remote branch, you might want to delete it:

git branch -d <branch-name>

If the branch has not been merged and you still want to delete it:

git branch -D <branch-name>

**Additional Commands**

* **List Remote Branches:**

To list all remote branches:

git branch -r

**Rebasing**

Rebasing in Git is a powerful feature that allows you to integrate changes from one branch into another, but in a different way compared to merging. It helps in maintaining a cleaner project history and can simplify the process of integrating changes. Here's a comprehensive guide on rebasing in Git:

**What is Rebasing?**

Rebasing is the process of moving or combining a sequence of commits to a new base commit. This effectively means replaying your changes on top of another branch. The result is a linear history with no merge commits, which can make the project history cleaner and easier to understand.

**Basic Rebasing Workflow**

1. **Switch to the Branch You Want to Rebase:**

First, check out the branch you want to rebase onto another branch:

git checkout <feature-branch>

1. **Rebase Onto the Target Branch:**

Rebase your current branch onto the target branch (usually the main branch):

git rebase <target-branch>

**For example**, to rebase feature-branch onto main:

**git checkout feature-branch**

**git rebase main**

1. **Resolve Conflicts (if any):**

If there are conflicts during the rebase, Git will pause and prompt you to resolve them:

* + Open the conflicting files and resolve the conflicts.
  + Stage the resolved files:

git add <resolved-file>

* + Continue the rebase:

git rebase –continue

* + If needed, you can abort the rebase process:

git rebase --abort

1. **Push the Rebased Branch:**

After a successful rebase, you need to force push the branch to update the remote repository because rebase rewrites history:

git push --force

Be cautious with force pushing, especially if other collaborators are working on the same branch, as it can overwrite their changes.

**Interactive Rebasing**

Interactive rebasing allows you to edit, squash, or reorder commits during the rebase process:

1. **Start an Interactive Rebase:**

Begin an interactive rebase from the commit you want to edit or reorder:

git rebase -i <commit-id>

Replace <commit-id> with the commit hash or the number of commits to rebase. For example, to rebase the last 3 commits:

git rebase -i HEAD~3

1. **Edit the Rebase To do List:**

An editor will open with a list of commits. You can change the commands to:

* + **pick:** Use the commit as is.
  + **reword:** Edit the commit message.
  + **edit:** Amend the commit.
  + **squash:** Combine the commit with the previous one.
  + **fixup:** Combine the commit with the previous one and discard the commit message.
  + **drop:** Remove the commit.

Make the desired changes and save the file.

1. **Complete the Rebase:**

Follow the prompts to resolve conflicts, edit commits, and continue the rebase process.

**Common Rebase Scenarios**

1. **Rebasing a Feature Branch:**

Rebasing a feature branch onto main before merging:

git checkout feature-branch

git rebase main

1. **Rebasing Multiple Commits:**

To clean up commit history before merging:

git rebase -i HEAD~5

Edit the list to squash or reorder commits as needed.

1. **Rebasing After Fetching:**

Update your branch with changes from the remote repository:

git fetch

git rebase origin/main

**Git workflows**

Git workflows are strategies for managing and integrating changes in a Git repository. They provide structured approaches to handle development, collaboration, and release processes. Here’s an overview of popular Git workflows and how they can be applied:

**1. Git Flow**

**Overview:** Git Flow is a well-defined workflow that organizes branches based on their purpose, making it suitable for larger projects with multiple release stages and environments.

**Key Branches:**

* **main (or master):** Contains the production-ready code.
* **develop:** Integrates features and fixes; the main branch for ongoing development.
* **Feature Branches:** Created from develop for new features or improvements.
* **Release Branches:** Created from develop for preparing releases, used for final tweaks before merging into main and develop.
* **Hotfix Branches:** Created from main to quickly address critical issues in production.

**Workflow:**

1. **Feature Development:**

git checkout -b feature/feature-name develop

1. **Release Preparation:**

git checkout -b release/v1.0 develop

1. **Hotfixes:**

git checkout -b hotfix/critical-issue main

**Advantages:**

* Clear structure with specific roles for each branch.
* Supports parallel development and multiple releases.

**Disadvantages:**

* Can be complex for small teams or simple projects.

**2. GitHub Flow**

**Overview:** GitHub Flow is a simpler workflow designed for continuous delivery. It’s often used in projects with frequent deployments.

**Key Branches:**

* **main (or master):** The stable branch that is always deployable.

**Workflow:**

1. **Create a Branch:**

git checkout -b branch-name

1. **Work and Commit:**

git add .

git commit -m "Description of changes"

1. **Push Branch:**

git push origin branch-name

1. **Create a Pull Request:** Open a pull request on GitHub to merge changes into main.
2. **Review and Merge:** Review and merge the pull request.

**Advantages:**

* Simple and streamlined.
* Encourages frequent, small changes and continuous deployment.

**Disadvantages:**

* Less structured compared to Git Flow.
* May not be suitable for complex release management.

**3. GitLab Flow**

**Overview:** GitLab Flow combines features of Git Flow and GitHub Flow, adding support for environments and issue tracking.

**Key Branches:**

* **main (or master):** Stable branch for production-ready code.
* **Environment Branches:** Branches for specific environments (e.g., staging, production).

**Workflow:**

1. **Feature Branches:** Develop features on separate branches.
2. **Environment Branches:** Create branches for staging or production.
3. **Merge Requests:** Use merge requests to integrate changes and deploy to specific environments.

**Advantages:**

* Flexible, supporting different deployment strategies.
* Integrates well with issue tracking and environments.

**Disadvantages:**

* Can become complex with multiple environments.
* Requires integration with issue tracking tools.

**4. Trunk-Based Development**

**Overview:** Trunk-Based Development focuses on having all developers work on a single branch (the "trunk" or main branch) with short-lived branches for features or fixes.

**Key Branches:**

* **main (or trunk):** The primary branch where all changes are integrated.

**Workflow:**

1. **Short-Lived Branches:** Create short-lived branches for features or fixes.
2. **Frequent Integration:** Regularly merge changes back into main to keep it deployable.

**Advantages:**

* Encourages continuous integration and frequent releases.
* Reduces merge conflicts by keeping branches short-lived.

**Disadvantages:**

* Requires discipline to keep changes small and frequent.
* Can be challenging for larger teams or complex projects.

**5. Feature Branch Workflow**

**Overview:** Feature Branch Workflow involves creating a new branch for each feature or task, making it easier to develop and test features in isolation.

**Key Branches:**

* **main (or master):** The stable branch.
* **Feature Branches:** Separate branches for each feature or task.

**Workflow:**

1. **Create a Feature Branch:**

git checkout -b feature/feature-name

1. **Develop and Commit Changes:**

git add .

git commit -m "Feature implementation"

1. **Merge Feature Branch:** Once the feature is complete, merge it into main or another integration branch.

**Advantages:**

* Isolates features, making development and testing easier.
* Provides clear history and context for each feature.

**Disadvantages:**

* Can lead to many branches, which need to be managed.
* Requires regular merging to avoid large divergences.