***GIT AND GITHUB***

**Git**

**Definition**: Git is a tool that helps you keep track of changes in your code. Imagine it as a save button that remembers every version of your project. Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It allows multiple developers to work on the same codebase simultaneously without interfering with each other's work.

**Key Points**:

* **Version Control**: It keeps a history of all changes, so you can go back to any previous version if needed.
* **Local**: You have a complete copy of your project on your computer.
* **Branches**: You can create different versions of your project to work on new features without affecting the main project.

**GitHub**

**Definition**: GitHub is a platform where you can store your Git projects online. It makes it easy to share your code with others and work together. GitHub provides a collaborative environment for developers. It offers additional features like bug tracking, task management, and social networking for programmers.

**Key Points**:

* **Repositories**: These are like folders where your projects are stored.
* **Collaboration**: You can invite others to work on your project, review each other's code, and discuss changes.
* **Extra Features**: GitHub offers tools to track bugs, manage tasks, and integrate with other services.
* Integration: Supports integration with various tools and services, enhancing the development workflow.

**Summary of Differences**

* **Purpose**: Git is a tool for version control, while GitHub is a platform for hosting Git repositories and facilitating collaboration.
* **Functionality**: Git operates locally on your machine, whereas GitHub provides a cloud-based interface for managing Git repositories.
* **Features**: Git focuses on version control, branching, and merging. GitHub adds features like issue tracking, pull requests, and social coding.

**Basic Linux/CMD Commands:**

ls - List all the files in the current directory / location. **$ ls**

mkdir - Make a new directory at your current location. **$ mkdir new\_directory**

cd – change directory (moves inside a file that exists in the current location**). $ cd new\_directory**

touch – create a new file **$ touch new\_file.txt**

pwd - Prints the current working directory $ pwd

/home/user/new\_directory

rm – removes a file or a directory $ rm new\_file.txt

cp – Copies a file or a directory $ cp <source> <destination>

cat – display the content of a file $ cat file\_name

**Version Control with Git**

**A VCS or the**[**Version Control System**](https://www.geeksforgeeks.org/version-control-systems/) is used to track versions of the files and store them in a specific place known as **repository**. The process of copying the content from an existing Git Repository with the help of various Git Tools is termed **git cloning.**Once the cloning process is done, the user gets the complete repository on his local machine. Git by default assumes the work to be done on the repository as a user, once the cloning is done. Users can also [create a new repository](https://www.geeksforgeeks.org/creating-repository-in-github/) or delete an existing repository. To delete a repository, the simpler way is to just delete the folder containing the repository.

**Working Directory**

The working directory in Git is the area where you make changes to your files. It's essentially your local copy of the project where you can edit, add, or delete files. Here are some key points about the working directory:

* **Local Changes**: Any modifications you make to files are done in the working directory. These changes are not yet tracked by Git until you add them to the staging area.
* **Untracked Files**: New files you create in the working directory are initially untracked by Git. You need to add them to the staging area to start tracking them.

**Staging Area**

The staging area in Git is a space where you can prepare changes before committing them to the repository. You add files to the staging area using the git add command. Once you're satisfied with the changes in the staging area, you can commit them to the repository with the git commit command. This helps you organize and control what gets included in each commit.

**Stashing Area**

The stashing area in Git is a temporary storage space where you can save your current changes without committing them. This allows you to switch tasks or branches without losing your progress. You can later retrieve and apply these saved changes when you're ready to continue working on them

Imagine you're working on a project, and you need to fix a bug in a different part of the code. However, you have some unfinished changes in your current work that you don't want to commit yet. Here's how stashing helps:

1. **Current Work**: You're editing files and making changes in your working directory.
2. **Need to Switch Tasks**: You suddenly need to switch to a different task, like fixing a bug.
3. **Stashing**: You use git stash to save your current changes. It's like putting your work in a temporary storage box.
4. **Clean State**: Your working directory is now clean, and you can switch to the bug-fixing task without losing your previous work.
5. **Returning to Your Work**: Once you're done with the bug fix, you can use git stash apply to bring back your saved changes from the storage box and continue working.

So, stashing is like putting your current work aside temporarily, allowing you to switch tasks and come back to your saved work later.

**Understanding the Working Tree in a Git Repository**

A working tree in a Git repository is a set of files from a specific version of the repository. It tracks changes made by a user to that version. When a user commits changes, Git only looks at the files in the working area, not all the modified files. Only the files in the working area are included in the commit. Users can change the files by editing, deleting, or adding new ones.

There are a few stages of a file in the working tree of a repository:

* **Untracked:** In this stage, the Git repository is unable to track the file, which means that the file is never staged, nor it is committed. The file is present in the working directory, but Git is unaware of its existence.
* **Tracked:** When the Git repository tracks a file, which means the file is committed but is not staged in the working directory. In this the file changes have been committed at some point in the repository’s history.
* **Staged:** In this stage, the file is ready to be committed and is placed in the staging area waiting for the next commit. The changes in the file have been marked and to be included in the next commit.
* **Modified/Dirty:** When the changes are made to the file i.e. the file is modified but the change is not yet staged.

After the changes are done in the working area, the user can either update these changes in the GIT repository or revert the changes

***GIT COMMANDS (basics)***

**1. git init**

* **Purpose**: Initializes a new Git repository.
* **Usage**: Run this command in your project directory to start tracking it with Git.
* **Example**: git init

**2. git status**

* **Purpose**: Shows the status of your working directory and staging area.
* **Usage**: Use this command to see which files are modified, staged, or untracked.
* **Example**: git status

**3. git add**

* **Purpose**: Adds changes from your working directory to the staging area.
* **Usage**: Use this command to prepare changes for a commit.
* **Example**: git add <filename> or git add . (to add all changes)

**4. git commit**

* **Purpose**: Commits the changes in the staging area to the repository.
* **Usage**: Use this command to save your changes with a message describing what you did.
* **Example**: git commit -m "Your commit message"

**5. git log**

* **Purpose**: Shows the commit history of the repository.
* **Usage**: Use this command to see a list of all commits made to the repository.
* **Example**: git log

**6. git stash**

* **Purpose**: Saves your current changes temporarily.
* **Usage**: Use this command to stash changes when you need to switch tasks.
* **Example**: git stash

**7. git stash apply**

* **Purpose**: Applies stashed changes back to your working directory.
* **Usage**: Use this command to retrieve and apply stashed changes.
* **Example**: git stash apply

**8. git clone**

* **Purpose**: Creates a copy of an existing Git repository.
* **Usage**: Use this command to download a repository from a remote server to your local machine.
* **Example**: git clone <repository\_url>

**git stash pop**

* **Purpose**: Applies the most recently stashed changes and removes them from the stash list.
* **Usage**: Use this command when you want to retrieve your changes and don't need to keep them in the stash anymore.
* **Example:** git stash pop

**git reset (soft, mixed, hard)**

* **Purpose**: Moves the HEAD pointer and optionally updates the working directory and staging area.
* **Usage**:
  + git reset --soft HEAD~1: Undo the last commit but keep changes staged.
  + git reset --mixed HEAD~1 (default): Undo the last commit and unstage the changes.
  + git reset --hard HEAD~1: Undo the last commit and delete the changes completely.
* **Tip**: Be careful with --hard, as it **deletes** your changes permanently.

**git revert**

* **Purpose**: Safely undo a specific commit by creating a new commit that reverses the changes.
* **Usage**: git revert <commit\_hash>

**git log --oneline**

* **Purpose**: Displays the commit history in a compact, single-line format for quick viewing.
* **Usage:** git log --oneline

***What is a Branch?***

A branch in Git is a separate line of development. It allows you to work on different features, fixes, or updates independently from the main codebase. Think of a branch as a copy of the project where you can make changes without affecting the main version.

**How are branches used:**

1. **Creating a Branch**: Use the git branch <branch\_name> command to create a new branch.
   * Example: git branch feature-xyz
2. **Switching to a Branch**: Use the git checkout <branch\_name> command to switch to the branch you want to work on.
   * Example: git checkout feature-xyz
3. **Making Changes**: Work on your changes in the new branch. These changes are isolated from the main branch.
4. **Committing Changes**: Use git add and git commit to save your changes in the branch.
5. **Merging Branches**: Once your work is complete, you can merge the branch back into the main branch using git merge <branch\_name>.
   * Example: git checkout main (switch to the main branch), git merge feature-xyz (merge the changes)
6. **Deleting a Branch**: After merging, you can delete the branch if it's no longer needed using git branch -d <branch\_name>.
   * Example: git branch -d feature-xyz

**GIT COMMANDS related to branches.**

**1. git branch**

* **Purpose**: Lists, creates, or deletes branches.
* **Usage**: Use this command to manage branches in your repository.
* **Example**: git branch (to list branches), git branch <branch\_name> (to create a new branch), git branch -d <branch\_name> (to delete a branch)

**2. git checkout**

* **Purpose**: Switches between branches or restores working directory files.
* **Usage**: Use this command to switch to a different branch or revert files to a previous state.
* **Example**: git checkout <branch\_name> (to switch branches), git checkout <commit\_hash> <filename> (to restore a file)

**3. git merge**

* **Purpose**: Combines changes from different branches.
* **Usage**: Use this command to merge changes from one branch into another.
* **Example**: git merge <branch\_name>

**4. git diff**

* **Purpose**: Shows differences between files.
* **Usage**: Use this command to see what changes have been made to files.
* **Example**: git diff

**What is a Remote Repository?**

A remote repository is a version of your project that is hosted on a server, typically on platforms like GitHub, GitLab, or Bitbucket. It allows multiple people to collaborate on the same project by sharing changes through a central location.

**How to Use Remote Repositories?**

1. **Cloning a Repository**:
   * **Purpose**: To create a local copy of a remote repository.
   * **Command**: git clone <repository\_url>
   * **Example**: git clone https://github.com/user/repo.git
2. **Adding a Remote**:
   * **Purpose**: To link your local repository to a remote repository.
   * **Command**: git remote add <remote\_name> <repository\_url>
   * **Example**: git remote add origin <https://github.com/user/repo.git>
3. **Fetching Changes**:
   * **Purpose**: To download changes from the remote repository without merging them.
   * **Command**: git fetch
   * **Example**: git fetch origin
4. **Pulling Changes**:
   * **Purpose**: To download and merge changes from the remote repository into your local branch.
   * **Command**: git pull
   * **Example**: git pull origin main
5. **Pushing Changes**:
   * **Purpose**: To upload your local commits to the remote repository.
   * **Command**: git push
   * **Example**: git push origin main
6. **Viewing Remote Repositories**:
   * **Purpose**: To list all remote repositories linked to your local repository.
   * **Command**: git remote -v
   * **Example**: git remote -v

**What is Merging?**

Merging is the process of combining changes from one branch into another. It's a way to integrate the work done in different branches, ensuring that all changes are included in the final version of the project.

**How to Merge Branches?**

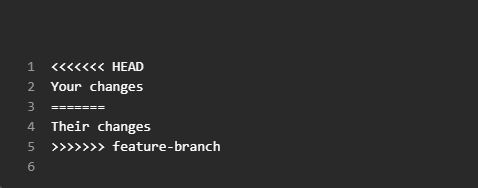
1. **Switch to the Target Branch**:
   * **Command**: git checkout <target\_branch>
   * **Example**: git checkout main
2. **Merge the Source Branch**:
   * **Command**: git merge <source\_branch>
   * **Example**: git merge feature-branch

**What are Merge Conflicts?**

Merge conflicts occur when Git cannot automatically resolve differences between branches. This usually happens when the same part of a file has been modified in different ways in the branches being merged.

**How to Resolve Merge Conflicts?**

1. **Identify Conflicts**:
   * Git will indicate which files have conflicts. You can use git status to see the list of conflicted files.
2. **Open Conflicted Files**:
   * Open the files with conflicts in your text editor. Conflicted sections will be marked with conflict markers:



**3 Resolve Conflicts**:

* + Edit the file to resolve the conflicts. Decide which changes to keep or combine them as needed.
  + Remove the conflict markers after resolving the conflicts.

**4 Add Resolved Files**:

* + **Command**: git add <filename>
  + **Example**: git add conflicted-file.txt

**5 Continue the Merge**:

* + **Command**: git commit (if the merge was paused) or git merge --continue (if using rebase)
  + **Example**: git commit (Git will automatically create a merge commit message)

**What is Rebasing?**

Rebasing is a way to integrate changes from one branch into another. It allows you to move or combine a series of commits to a new base commit. Unlike merging, which creates a new commit to combine changes, rebasing rewrites the commit history.

**How to Use Rebasing?**

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

* + Example: git checkout feature-branch

1. **Rebase Onto Another Branch**:
   * Example: git rebase main
   * This command moves the commits from feature-branch to the tip of the main branch.
2. **Resolve Conflicts** (if any):
   * If there are conflicts, Git will pause the rebase and ask you to resolve them. You can use git status to see which files have conflicts.
   * Resolve the conflicts manually, then use git add <filename> to mark them as resolved.
   * Continue the rebase with git rebase --continue.
3. **Abort Rebase** (if needed):
   * If you want to stop the rebase process, you can use git rebase --abort to return to the state before the rebase started.

**What is Cherry-Picking?**

Cherry-picking in Git is the process of selecting specific commits from one branch and applying them to another branch. This allows you to pick and choose individual changes without merging entire branches.

**Why Use Cherry-Picking?**

Cherry-picking is useful for several reasons:

1. **Selective Integration**: It allows you to apply specific changes without bringing in all the changes from a branch.
2. **Bug Fixes**: You can quickly apply bug fixes from one branch to another without merging unrelated changes.
3. **Feature Isolation**: It helps in isolating and applying specific features or updates.

**How to Use Cherry-Picking?**

Here are the steps to cherry-pick a commit:

1. **Identify the Commit**:
   * Find the commit hash of the commit you want to cherry-pick. You can use git log to view the commit history.
   * Example: git log
2. **Switch to the Target Branch**:
   * **Command**: git checkout <target\_branch>
   * **Example**: git checkout main
3. **Cherry-Pick the Commit**:
   * **Command**: git cherry-pick <commit\_hash>
   * **Example**: git cherry-pick abc1234

**Resolving Conflicts During Cherry-Picking:**

If there are conflicts during cherry-picking, Git will pause the process and ask you to resolve them:

1. **Identify Conflicts**:
   * Git will indicate which files have conflicts. Use git status to see the list of conflicted files.
2. **Open Conflicted Files**:
   * Open the files with conflicts in your text editor. Conflicted sections will be marked with conflict markers.
3. **Resolve Conflicts**:
   * Edit the file to resolve the conflicts. Decide which changes to keep or combine them as needed.
   * Remove the conflict markers after resolving the conflicts.
4. **Add Resolved Files**:
   * **Command**: git add <filename>
   * **Example**: git add conflicted-file.txt
5. **Continue Cherry-Picking**:
   * **Command**: git cherry-pick --continue
   * **Example**: git cherry-pick --continue

**What is Forking?**

Forking is the process of creating a copy of a repository under your own GitHub account. This allows you to make changes to the project independently without affecting the original repository. Forking is commonly used to contribute to open-source projects.

**How to Fork a Repository?**

Here are the steps to fork a repository:

1. **Navigate to the Repository**:
   * Go to the GitHub page of the repository you want to fork.
2. **Fork the Repository**:
   * Click the "Fork" button at the top right corner of the repository page.
   * This creates a copy of the repository under your GitHub account.

**Fork vs Clone**

* **Fork**: Creates a full copy of a repository **under your GitHub account**. Useful for contributing to others' projects.
* **Clone**: Creates a local copy of a repository **on your machine**. Useful to work locally on a repository (your own or forked).

**What is a Pull Request?**

A pull request is a way to propose changes to a repository. After forking a repository and making changes, you can create a pull request to submit your changes for review. The repository owner can then review, discuss, and merge your changes.

**Pull Request**

* After forking and cloning, and making your changes:
  + Create a Pull Request (PR).
  + The repository owners and team members will **review** your changes.
  + Reviewers may **approve** the PR, **request changes**, or **comment**.
  + Once approved, the PR can be merged into the main repository.
  + Optionally, you can choose "**Squash and Merge**" to combine multiple commits into one before merging.