Git is a distributed version control system that helps track changes in files, particularly in software development projects. It allows multiple developers to collaborate on a project by managing different versions of files, making it easier to work on the same project without overwriting each other's work.

When I say Git is **distributed**, I mean that every developer working with Git has a **full copy of the entire repository**, including its complete history, on their own machine. This is in contrast to **centralized version control systems** (like SVN), where developers work on a single, central repository stored on a server, and only the server holds the full history of the project.

**Git Merge**

git merge is a fundamental Git command that integrates changes from one branch into another. It allows you to combine the history of two branches.

**Types of Merges in Git**

1. **Fast-Forward Merge**
2. **Three-Way Merge**

**1. Fast-Forward Merge**

This happens when the current branch (the one you're merging into) is directly behind the branch you're merging. In this case, Git doesn't need to create a new merge commit; it simply moves the pointer of the current branch forward to the other branch's tip.

**2. Three-Way Merge**

This occurs when the branches have diverged. In this case, Git looks at the common ancestor of both branches (the merge base), then compares the changes in the two branches since that point and attempts to reconcile the differences.

**Merge Conflicts**

A merge conflict occurs when Git can't automatically reconcile differences between two branches. This typically happens when the same lines of the same file have been changed in both branches in incompatible ways. Git will mark the file as conflicted, and you'll need to manually resolve the conflict before completing the merge.

**Merge Commit**

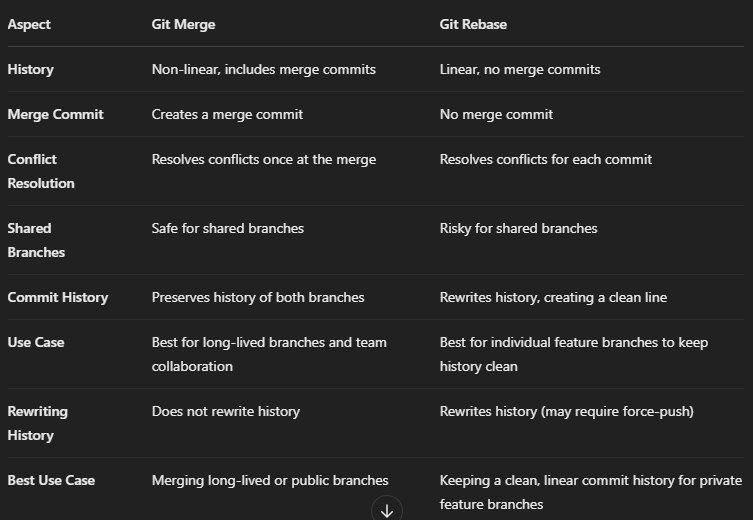
Once the merge is complete (either through fast-forward or three-way merge), Git creates a **merge commit** to represent the integration of the changes. The merge commit has two parent commits, showing the history of both branches.

**Common Merge Options**

1. **–-no-ff:** By default, Git will fast-forward if possible. However, using the --no-ff option forces a merge commit even if a fast-forward merge is possible.
2. **–-squash:** This option is used to combine all the changes from a branch into a single commit, rather than keeping the individual commits from the feature branch.
3. **--abort**: If you start a merge and realize something went wrong (e.g., a complex conflict), you can abort the merge and return to the state before the merge started.
4. **--no-commit**: This option prevents Git from automatically committing after the merge, allowing you to review the changes before committing.

git rebase is a powerful Git command that integrates changes from one branch into another. Unlike git merge, which creates a new merge commit, git rebase rewrites the commit history, applying changes from one branch onto another. The result is a linear history, as opposed to the branching history that occurs with merges.

Rebasing is commonly used to maintain a cleaner, linear project history, especially in workflows where developers want to avoid merge commits.



**Git reset**

The git reset command in Git is used to undo changes in the working directory and staging area.

**1. Soft Reset (git reset --soft <commit>)**

A **soft reset** moves the HEAD pointer to the specified commit, but it **does not touch the staging area (index) or working directory**. This means that all your changes remain in the staging area, and you can commit them again or modify them further.

**2. Mixed Reset (git reset --mixed <commit>) (default)**

A **mixed reset** is the default behavior of git reset. It moves the HEAD pointer to the specified commit, resets the staging area (index) to match the commit, but **does not modify the working directory**. This means your changes are **removed from the staging area** but are **kept in your working directory**.

**3. Hard Reset (git reset --hard <commit>)**

A **hard reset** moves the HEAD pointer to the specified commit and **clears both the staging area and the working directory** to match the commit. This means that all changes from the commits after the specified commit are **discarded**, and your working directory is reverted to exactly match that commit.

**Git restore**

git restore is a relatively newer Git command introduced to simplify the process of undoing changes in your working directory and staging area. It is often seen as an alternative to certain uses of git checkout and git reset.

**Key Uses of git restore**

1. Restoring a File to the Last Committed State
2. Restoring Staged Changes (Unstaging Files)
3. Restoring Files from Another Commit
4. Restoring All Files in the Working Directory

**Git revert**

git revert is a Git command used to **undo** the effects of a specific commit by creating a **new commit** that reverses the changes introduced by the original commit. Unlike git reset, which alters the commit history, git revert creates a new commit to **preserve the history** while undoing a specific commit's changes.

* 1. **Preserves history**: Unlike git reset, git revert doesn’t rewrite history. It simply creates a new commit that undoes the changes made by a previous commit.
  2. **Safe for shared repositories**: Because git revert doesn’t change the commit history, it is a **safe way** to undo changes in a collaborative or shared repository without disrupting other contributors.
  3. **Reverts individual commits**: You can specify a single commit to revert, and Git will generate a new commit that undoes it.

**Key Uses of git checkout**

1. Switching Between Branches
2. Creating and Switching to a New Branch
3. Restoring Files (Reverting Changes)
4. Checking Out Specific Commits
5. Detaching HEAD
6. Checking Out a File from Another Branch

**Git tag**

The **git tag** command in Git is used to create, list, delete, or verify tags. Tags are references that point to specific points in Git history, typically used to mark release points (e.g., v1.0, v2.0).

Git tag

git tag -a v1.0 -m "Release version 1.0" 🡪 annotated tag

**Git cherry-pick**

The **git cherry-pick** command is used to apply the changes introduced by some existing commits onto the current branch. This is useful when you want to copy specific commits from one branch to another without merging the entire branch.

Git cherry-pick <commit-hash>

A black background with white text and red circles

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Git stash**

In **Git**, git stash is a command used to temporarily save changes that are not yet committed, so you can switch to another branch or work on something else without losing your current work. The stashed changes are stored in a stack-like structure and can be reapplied later.

Git stash 🡪 To save your changes (both staged and unstaged) without committing them, you can use

git stash save "message describing the stash" 🡪 saving with a message

git stash apply 🡪 To reapply the most recent stash

git branch

git branch -r

git branch -a

git branch “new\_branch”

git branch -m <new-branch-name>

git branch -d <branch-name>

git commit –amed -m “new-commit-message”