1. GIT LOG:

The git log command shows the **commit history** of a Git repository. It allows you to see all the commits that have been made, along with details like:

1. Commit hash (ID)
2. Author name and email
3. Date and time
4. Commit message

🔧 Commonly Used git log Options:

Here are some **powerful options** you can use with **git log.**

1. ***--oneline:*** Shows each commit in one line (compact format).
2. ***--graph:*** Displays an ASCII graph/tree of branches and merges.

   You can also combine it with --oneline:

   - ***git log --oneline --graph –all***
3. ***--author*** **="name":** Show commits by a specific author.
4. ***--since and --until:*** Filter commits by date range.
5. ***--stat:*** Show file changes (stats) for each commit.
6. ***-p (patch):*** Shows the actual changes (diffs) introduced in each commit.
7. ***-n or --max-count:*** Limit the number of commits.

   - ***git log -3***

🗂️ Example: Full Power Combo

git log --oneline --graph --decorate --all

- --decorate: shows branch and tag names
- --all: includes logs from all branches

📁 git log <file>

- Shows commit history of a **specific file**:

📜 What is git show?

The git show command in Git is used to display detailed information about:

- A specific commit
- A file change
- A tag
- A branch

It shows:

- The commit message
- The author
- The date
- And the actual code changes (diff) introduced in that commit

2. UNDOING CHANGES(RESTORE, RESET)

- **Case 1: Undo Unstaged Changes (File edited, but not added)**
  - ➢ *git restore <file>*

- **Case 2: Undo Staged Changes (Added with git add, but not committed)**
  - ➢ *git restore --staged <file>*

- **Case 3: You just made a commit, but want to undo it?**

  1. **Undo the commit, keep your code:**

  ➢ *git reset --soft HEAD~1*

  # This removes the last commit, but keeps all your code and staged changes.

  2. **Undo the commit, keep code but unstage it:**

  ➢ *git reset --mixed HEAD~1*

  # Removes the last commit and unstages the changes.

  3. **Undo the commit and delete the code changes:**

  ➢ *git reset --hard HEAD~1*

  ⚠️ Warning: This deletes your changes permanently!

3. REVERT:

creates a new commit that undoes the changes made by a previous commit — without deleting any history.

- **You already pushed the commit and want to undo it?**

  # Don't delete! Create a new "undo" commit:

  ➢ ***git revert <commit-id>***

  # Find the commit ID using: **git log**

Commit	Change	file.txt Contents
A	Added "Hello"	Hello
B	Added "This is a second line"	Hello + 2nd line
C	Reverted commit B	Back to just Hello

- **"*git commit --amend* is ideal for fixing your latest local commit before pushing."**

- **"*git revert HEAD* is ideal for undoing a commit after you've pushed it, without breaking the shared history."**

Revert a Specific Commit: ***git revert <commit id>***

4. GIT BRANCH:

The git branch command is used to manage branches in Git.

When you **change a branch in Git**, it's like switching to a different line of work — each branch has its **own timeline (commits, changes, and history)**.

- **Basic Usage:**
  - ***git branch*** # Shows all branches
  - ***git branch <name>*** # Creates a new branch
  - ***git branch -m <name>*** # to rename branch
  - ***git branch -d <name>*** # Deletes a branch from local repo
  - ***git branch -D <name>*** # forces a branch to be deleted.
  - ***git push origin --delete <branch-name>*** # delete a branch from remote
  - ***git checkout <branch name>*** # to navigate
  - ***git checkout -b <new branch name>*** # to create new branch and switch to that new branch (this is shortcut)

[Note: if you are in the main branch you can't delete main branch, first you need to go to other branch and than you can delete the main branch.]

5. GIT MERGING:

🧠 **Conceptual Diagram**

Before Merge:

main: A---B---C

****

feature: D---E

After: (if merged on main)

main: A---B---C--------F (merge commit)

\ /

feature: D---E

- **Basic Syntax:**
  - ***git merge <branch-name>***

Step-by-step:

You are on main

git checkout main

You want to bring changes from 'feature' into 'main'

git merge feature

🔹 **Fast-Forward merge:**

A Fast-Forward merge happens when the current branch has no new commits since it split from the branch you're merging in.

So Git can just move the pointer forward — **no need** to create a **new merge commit.**

Visual Example (Before Merge):

main: **A---B**

\

feature: **C---D**

- You are on main.
- feature has 2 new commits: C and D.
- main has not changed since the branch.

Command:

git checkout main

git merge feature

Result: Fast-Forward Merge(After Merge):

Git sees that main is behind and just moves it forward to the latest commit of feature:

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

feature: ↑

👉 No new merge commit is created.

- **Three - Way merge:**

A Three-Way Merge happens when:

- Both branches have made new commits since they split.
- Git can't fast-forward, so it creates a merge commit to combine the changes.

Visual Example(Before Merge):

Suppose this is your Git history:

Main: **A---B---C---C1**

\

\

Feature: **D1---D2**

- C is the common ancestor.
- C1 is the latest on main.
- D2 is the latest on feature.

Command:

git checkout main

git merge feature

Result: three-way merge(After Merge):

A---B---C ---C1 ----- M

\ /

\ /

D1---D2 (feature)

M = the merge commit that combines main and feature. We don't need to create **M**. git creates by itself.

NOTE: ***git merge --abort*** **cancels an in-progress merge and restores your branch to the exact state it was before you ran git merge.**

🔄 When to Use:

- Merge has conflicts
- You haven't committed yet
- You want to cancel the merge

❗ Key Points:

Situation	Can Use --abort?
Conflicts during merge	✅ Yes
Merge completed (no conflict)	❌ No
Conflicts resolved + committed	❌ No

6. GIT REBASE:

git rebase is a Git command that moves or reapplies commits from one branch on top of another.
It helps keep your commit history clean, linear, and readable.

- **Use rebase to clean history before merging**

✅ Clean History — no messy merge commits

✅ Linear Timeline — looks like one straight story of development

✅ Easier Debugging — simpler git log, git bisect

✅ Better Collaboration — makes pull requests cleaner

🔹 **Basic Syntax:**

#Rebase your branch onto main

git checkout feature

git rebase main

Visual Example(Before Rebase):

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

\

feature: **D---E**

- A, B, C are commits on main
- D, E are your commits on feature
- Now main has new commit C, and your branch is behind.

Command:

git checkout feature

git rebase main

What Happens During Rebase:

1. Git temporarily removes your commits D and E.
2. Git moves your branch to the tip of main (after C)
3. Git re-applies D and E on top of C.

Result: After Rebase:

main: A---B---C

\

feature: D'---E'

- D' and E' are new copies of your commits (with new commit IDs).
- History is now linear and clean.
- No merge commit is created.

Before (Branch History Looks Like Tree)

A---B---C (main)

\

D---E (feature)

After Rebase (Linear History)

A---B---C---D'---E' (feature, rebased)

⚠️ **Important:**

- Rebasing rewrites commit history
- Don't rebase a branch that you've already pushed to remote (unless you're okay to --force push)

Common Variants:

Command	Description
git rebase <branch>	Rebase your current branch onto another branch
git rebase -i <commit>	Start interactive rebase from a certain commit (great for squashing, editing)
git rebase --continue	Continue rebase after resolving conflicts
git rebase --abort	Cancel the rebase process and return to original state
git rebase --skip	Skip the commit causing conflict and continue
git rebase origin/main	Rebase on top of the remote main branch