Git Basic Commands

1. `git clone <repository URL>`: Downloads a code repository from the specified URL.
2. `git status`: Shows the current state of your local repository (modified files, staged changes, untracked files).
3. `git add <file>`: Adds a specific file to the staging area for the next commit.
4. `git commit -a -m "Message"`: Commits all modified and staged changes with a message.
5. `git commit -m "My message"`: Commits only staged changes with a custom message.
6. `git commit --amend -m "My new amended message for last commit"`: Edits the message of the most recent commit.
7. `git log -number` Where number indicates the number of commits back you want to see.

How to Revert a Commit in Git

Undo a mistake with a new "opposite" commit.

7. `git revert <commit>`: Reverts the specified commit using its SHA or branch name with offset (~).
8. `git revert --no-edit <commit>`: Same as above, but skips editing the commit message.
9. `git revert HEAD~5..HEAD~2`: Reverts a range of commits between two revisions.
10. `git revert -n <commit>`: Reverts a commit but allows further changes before committing.

How to Configure Branches in Git

11. `git branch <name of branch>`: This will create a Git branch on your currently checked-out reference.
12. `git branch -r <name of branch>`: This will list all remote branches.

13. `git push -u` This will delete reflect the changes of the local Git branch on remote branch.
14. `git checkout <name-of-branch>` This will update your repository's files to match the snapshot of the commit you have checked out. This can be great for reviewing old commits without creating a new branch.
15. `git checkout <commit hash>` This will update your repository's files to match the snapshot of the commit you have checked out. This can be great for reviewing old commits without creating a new branch.

It's important to be aware: checking out a Git commit will move the HEAD point to a commit instead of a branch. This will put your repository in what's called detached HEAD state. When you are in detached HEAD state in Git, you are not able to commit changes to a branch. To exit detached head state, checkout a branch.

16. `git checkout -b <branch name>` Saves time by creating branch and checking it out at the same time.
17. `git branch -m <old branch name> <new branch name>` Renames branch without checking it out.

Note: one cannot delete a local branch if it's checked out.

18. `git branch -d <branch name>` Delete LOCAL branch with merged changes.
19. `git branch -D <branch name>` Delete LOCAL branch with UN-merged changes.
20. `git push origin --delete <branch name>` Delete REMOTE branch.

Merging: is used to combine changes from one branch to another, such as Git merge to main.

1. `git status` first, check the current state of your repo
2. `git commit -m "commit message"` create a commit with a description.
3. `git checkout <branch name>` switch to the branch where you want to integrate the changes from your feature branch.

4. `git pull` get the latest version of the target branch (optional)
5. `git merge <name of the feature branch>` merge the feature you wish to integrate.

Rebasing: is an action in Git that allows you to rewrite commits from one Git branch to another branch. Essentially, Git rebase is deleting commits from one branch and adding them to another.

To rebase first checkout the branch that's getting moved in the rebase. Then tell Git what branch to rewrite the commits onto.

1. `git status` first, check the current state of your repo
2. `git commit -m "commit message"` create a commit with a description.
3. `git checkout <feature branch>` switch to the branch where you want to integrate the changes from your feature branch.
4. `git log -#` view the most recent commits on your feature branch. This helps you understand where your feature branch originally diverged from the target branch.
5. `git pull` get the latest version of the target branch (optional)
6. `git rebase <target branch>` This replays your feature branch commits on top of the latest commit in the target branch.

Git pull vs. fetch:

- **Git fetch:** (`git fetch`) Downloads changes (commits and branches) from the remote repository **without** modifying your local working directory. Useful for:
    - Checking for remote updates before making changes.
    - Avoiding conflicts when merging later.
- **Git pull:** (`git pull`) Combines `fetch` with a `merge`, incorporating remote changes **directly** into your local branch. Use when:
    - You're confident about remote changes and want them applied right away.

    - Convenience matters (faster than separate `fetch` and `merge`).

In short:

- **Fetch for awareness:** See what's changed remotely without affecting your local work.
- **Pull for action:** Get and apply remote changes directly to your local branch.

Git Stash

- **Stashing**: Temporarily saves uncommitted changes for later use.
- Useful for switching tasks or collaborating without losing progress.

Applying Stash (2 Options):

1. **git stash apply**: Applies changes from the latest stash, but keeps the stash for future use.
2. **git stash pop**: Applies changes from the latest stash and removes it.

Viewing Stash List:

- Use `git stash list` to see all stashes with their corresponding index.
- Use `git stash show stash@{index}` to view the contents of a specific stash (replace `{index}` with the actual index number).

Cherry-picking vs. Squashing in Git

- Both cherry-picking and squashing are used to incorporate changes from one branch (source) into another (target).
- They differ in how they achieve this.

Cherry-picking (One Commit at a Time):

- Picks individual commits from the source branch and applies them to the target branch.
- Creates new commits in the target branch history for each picked commit.
- Useful for incorporating specific changes without the entire source branch.
- Use cherry-picking for incorporating specific changes or maintaining detailed history.

Squashing (Merge & Rewrite):

- Merges all changes from the source branch into the target branch.
- Creates a single new commit in the target branch history summarizing the merged changes.
- Useful for keeping a cleaner branch history and hiding intermediate steps.
- Use squashing for a cleaner history or when the entire source branch is no longer needed.

Remember: Squashing rewrites history, use it with caution, especially in shared repositories.

Git Tags

- **Function:** References to specific commits in a Git repository history.
- **Common Uses:**
  - Marking releases (e.g., v1.0.0)
  - Referencing specific points in development without using the full commit hash

Creating Git Tags:

- **Lightweight Tag (Simple Reference):**
  - Command: `git tag <tag-name>` (e.g., `git tag my-release`)
- **Annotated Tag (Recommended - Includes Message):**
  - Command: `git tag -a <tag-name> -m "<message>"` (e.g., `git tag -a v1.0.0 -m "First stable release"`)

Verifying Tag Information:

- Command: `git show <tag-name>` (Displays tag info including message)

Pushing Tags:

- Push a specific tag: `git push origin <tag-name>` (e.g., `git push origin v1.0.0`)
- Push all local tags: `git push origin --tags`

Git Reset:

- Moves your working directory and/or index to a specific state in your Git history.
- **Essentially "undoes" changes** but with varying degrees of permanence.

Types of Reset:

1. **Soft Reset (`git reset --soft <commit>`):**
    - Most lightweight.
    - Moves HEAD (reference to latest commit) to a specific commit.
    - **Changes remain STAGED (in index) but not committed.**
    - Useful for amending the last commit message or adding forgotten files.
2. **Mixed Reset (`git reset --mixed <commit>`):**
    - Moves HEAD and discards all **committed changes** after the specified commit.
    - Changes **remain UNSTAGED (in index)** that were part of the discarded commits.
    - Use with caution as it rewrites history.
3. **Hard Reset (`git reset --hard <commit>`):**
    - Most drastic.
    - Moves HEAD, **discards** all committed and staged changes after the specified commit.
    - Working directory reflects the state of the chosen commit.
    - **Use only if confident in discarding local changes!**

Additional Options:

- `<commit>` can be replaced with relative references like `HEAD~1` (one commit back).
- `git reset <file>`: Unstages specific changes in the index.

What is Git LFS?

Git LFS (Large File Storage) is an open-source extension for Git that helps manage large files within your version control system. It overcomes limitations of storing large files directly in Git repositories, which can slow down performance and consume storage space.

The reason is that git can track changes in text files easily but, tracking changes in binary data such as images, audio, and video, means generating a new copy every time a change is made. This makes the local and remote repos large and slow. Any time you checkout a branch in the LFS repo you only pull down the file version you need.

Git Patches:

- Git patches were used to share code between developers.
- Patches are text files containing code changes and Git commit info.
- Some large projects (Linux Kernel, Drupal) still use them.

Creating a Git Patch (git format-patch):

- For a single commit: `git format-patch -1 <commit SHA> -o <output directory>`
- For a branch: `git format-patch <branch name> -o <output directory>`
- Refer to Git documentation for more options.

Applying a Git Patch (git apply):

1. Checkout the desired commit/branch.
2. Run: `git apply <.patch file>`

What is a Git Submodule?

- Like a child repository of one parent repository
- Pointer commits need to be manually updated.
- Useful for working in parallel.

Git Worktree:

Git worktree lets you checkout and work on multiple Git branches at the same time within separate directories.

Why use it?

- Avoid stashing and switching branches for quick tasks on different branches.
- Isolate work on a feature branch while keeping your main branch development ongoing.

Key Concepts:

- Each worktree points to a specific directory and a branch.
- Use `git worktree add` to create new worktrees for branches. (**Details below**).
- Use `git worktree list` to see active worktrees.
- Use `git worktree remove` to delete worktrees (use with caution!).

Common Use Case:

- Working on a feature branch while needing to test changes in another branch.
    - Checkout the feature branch in its own directory using `git worktree add`.
    - Work on the feature branch in its directory.
    - Switch to the other branch's directory to test changes.
    - Return to the feature branch directory to resume work.

Git Worktree Add Details:

- Creates a new worktree for a specified branch in a chosen directory.
- Offers flexibility in naming the directory:
    - Same name as the branch (e.g., `git worktree add feature features`)
    - Different name from the branch (e.g., `git worktree add feature my-feature-work`)
- Can also create a new branch and worktree simultaneously.

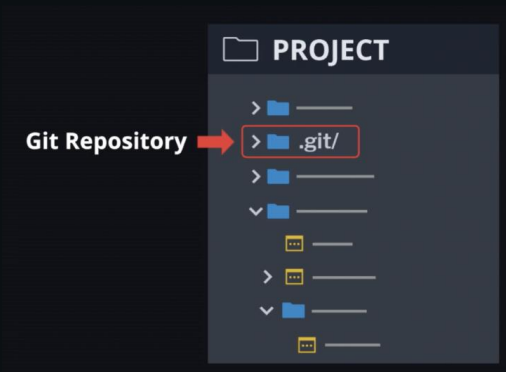
Additional Notes:

- New worktrees can share the same name as the branch or have a custom name.
- Refer to official Git documentation for more advanced usage.

Git repository

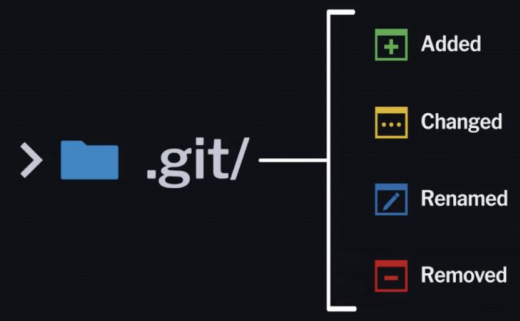
The .git/ folder inside a project.

Tracks all changes made to your project over time.



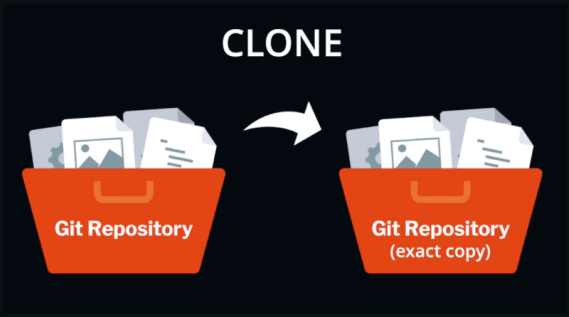
Git Init

The Git command that creates the .git/ folder in a project.



Git Clone

Creates an exact copy of all files and changes in a Git repo at the time of the clone.

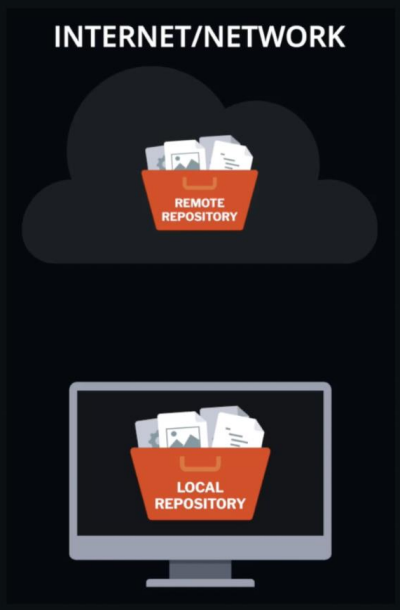


Remote Repository

A Git repository hosted on the internet or network.

Local Repository

A Git repository hosted on your machine.

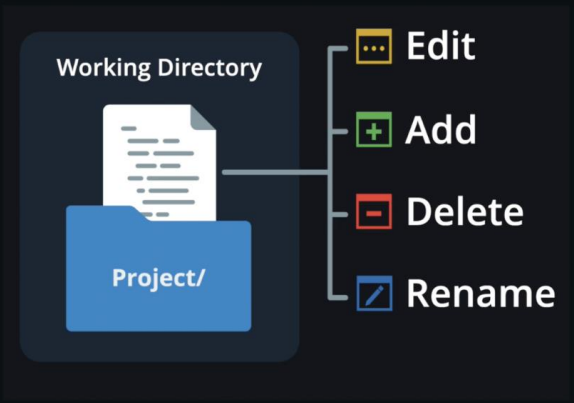


Commit

A snapshot of your repository at one point in time.

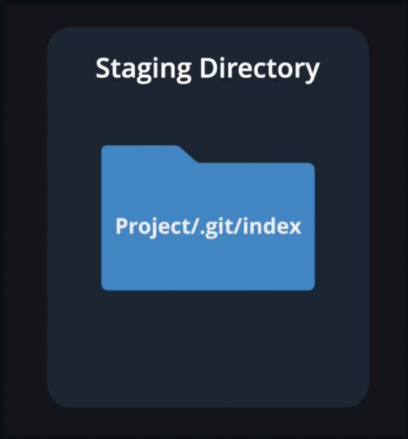
Working Directory

The file system where you can view and modify files.



Staging Directory

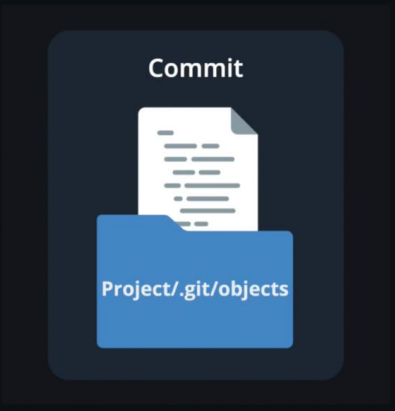
The `/index` folder inside the `.git/` folder that contains the changes added through staging.



Commit

Commits are stored in the `/objects` folder in the `.git/` folder.

Each commit has a unique ID number or “SHA.”

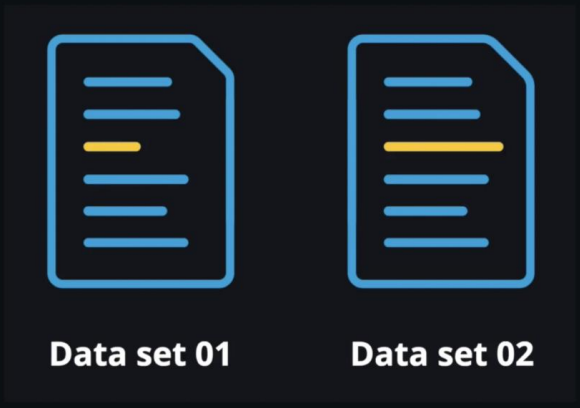


Typical Git Workflow

1) Make changes
2) Stage changes
3) Commit changes

Diff

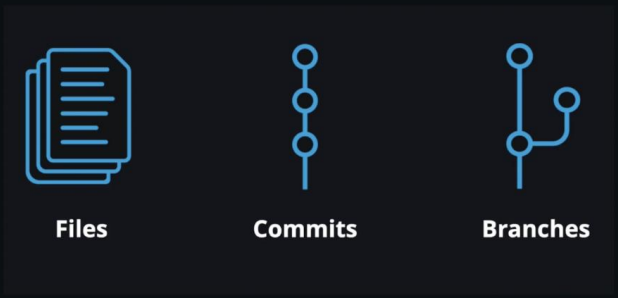
A change between 2 data sets.



Diff

A change between 2 data sets.

Commonly used between files, commits, and branches.



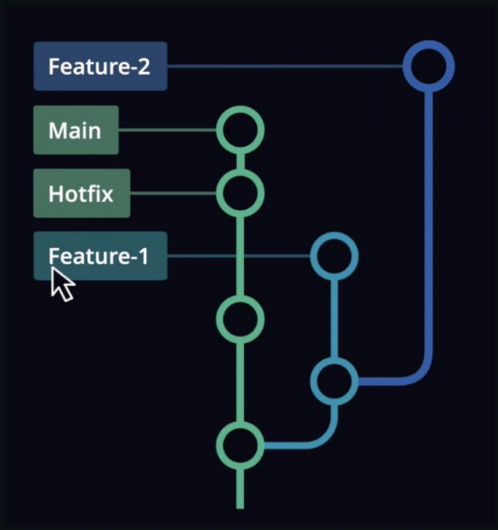
Any two data sets imaginable: think images, text, CSVs etc

Branch

A pointer to a specific commit.

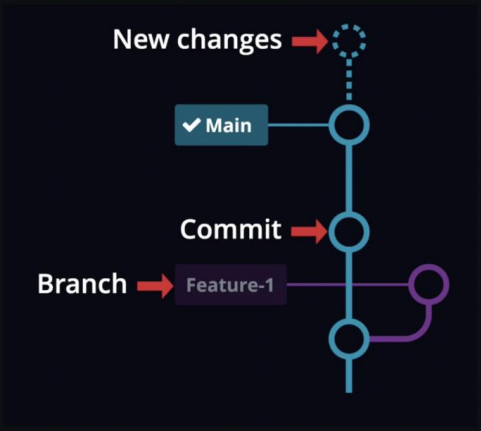
Useful for:

- Organizing work
- Building the foundation for
  - Merge
  - Rebase
  - And much more



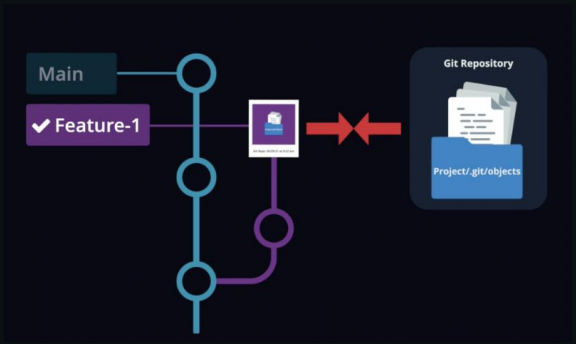
Git Checkout

Tells Git which branch or commit to apply new changes.



Git Checkout

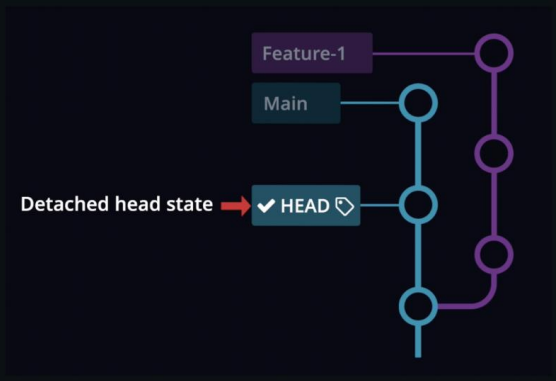
Updates your repo's file to match snapshot of whichever commit the branch points to



Detached HEAD State

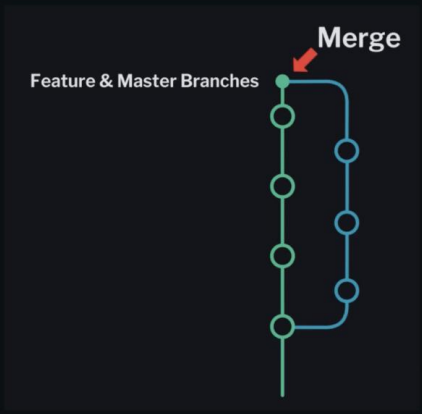
Checking out a commit moves the HEAD pointer to a commit instead of branch.

⚠️ Any changes commits will not be committed on a branch.



Merge

Adds all changes from one branch into another branch.



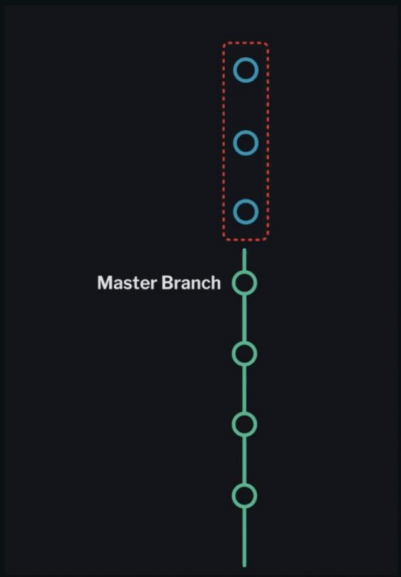
Merging:

- preserves history
- better for merge conflicts
- easy to undo

Rebase

Rewrites the commits from one branch and places them onto another branch.

This changes the tree structure by moving the commits and their changes onto the target branch.

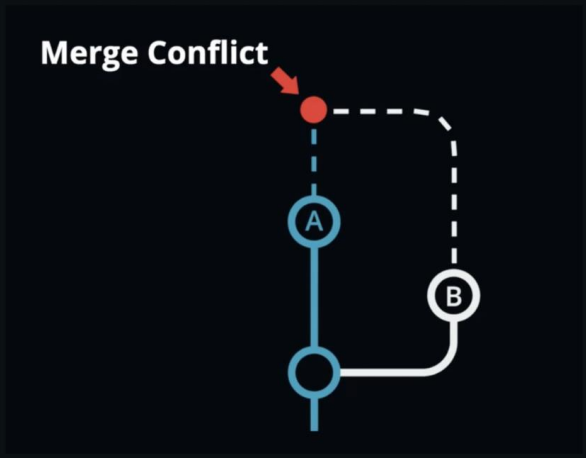


Rebasing:

- cleaner history
- more readable graph
- tougher to resolve conflicts

Merge Conflict

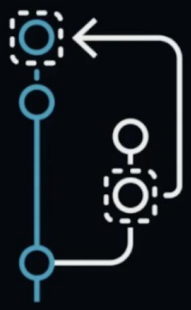
An event triggered when Git is unable to automatically resolve differences in code between two commits.



Conflicts may trigger when:

Merging a branch

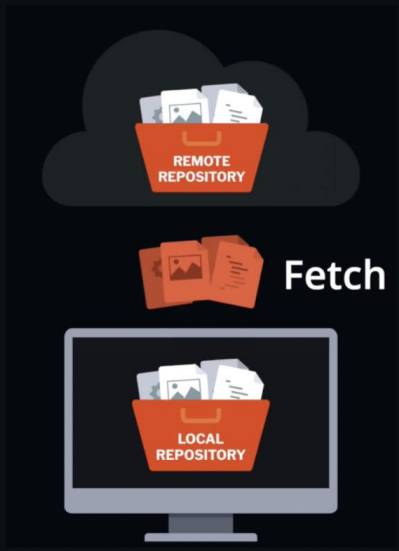
Rebasing a branch



Cherry picking

Git Fetch

Downloads all the changes from your remote to your local repo



Git Pull

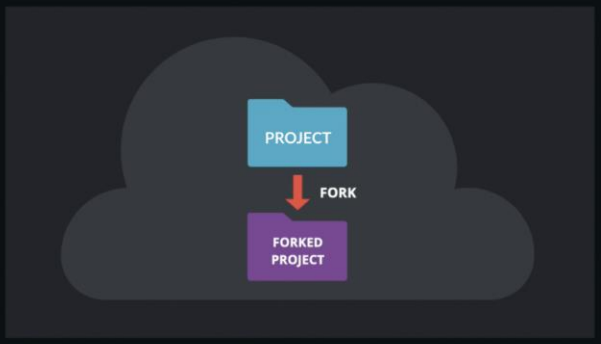
Updates your local clone of a Git repository with any changes from the remote repository.

Represents a combination of 2 actions:

- Git Fetch
- Git Merge

Fork

A clone of a git repository, typically hosted on a git hosting service.



Pull Request

An event where a contributor asks a repo maintainer to review code that they wish to merge into a project.

Also known as a merge request.

SSH

A network protocol that allows one computer to securely connect to another computer over an unsecured network, like the internet.

Encrypts your data so that you may:

- Securely log in to a remote machine
- Securely transmit files
- Or safely issue remote commands and more