Git Interview Preparation

Part 1: The Need for Version Control

1. How did teams manage code before Git ?

Answer: Before Distributed systems like Git, teams primarily used two methods:

1. **Manual Sharing:** Zipping code folders and sharing them via email, FTP, or shared network drives (e.g., `project_v1`, `project_final`).
2. **Centralized Version Control System (CVCS):** Tools like SVN (Subversion) or CVS, where code is stored on a single central server and developers must connect to it to download and save changes.

- **Simple Explanation:** It was like working on a group project by emailing a Word document back and forth. You had to wait for your turn to edit.
- **Real-World Example:** In the old days, a developer might tell the team, “I’m editing `login.php`; nobody else should modify it right now,” to prevent conflicts.

2. What problems existed in traditional (non-Git) code management ?

Answer: The traditional centralized model had significant flaws:

1. **Single Point of Failure:** If the central server crashed or the network went down, development stopped completely. No one could save work or look at history.
2. **No Offline Access:** You could not commit changes or switch branches without an internet connection.
3. **Slower Performance:** Every command required a network request to the central server."

- **Analogy:** It’s like a library where you are not allowed to take books home. If the library closes (server down), you can't read (work).

3. What kind of issues occurred without Git ?

Answer: Without a robust system like Git, teams faced:

- **Overwritten Work:** If two developers edited the same file and uploaded it via FTP, the second upload would overwrite the first, deleting code without warning.

- **Lack of Accountability:** It was difficult to trace exactly *who* broke the code and *when*.
- **Version Hell:** Folders became messy with names like `code_backup`, `code_new`, `code_final_real`.
- **Real-World Scenario:** A developer fixes a bug on the live server directly. The next day, another developer uploads an older version of the file from their computer, accidentally bringing the bug back.

4. How did Git solve these problems ?

Answer: Git introduced a **Distributed** model:

1. **No Single Point of Failure:** Every developer has a *full copy* of the entire project history on their local machine. If the server dies, any computer can restore it.
2. **Offline Capability:** You can commit, branch, and view logs while disconnected (e.g., on a plane).
3. **Data Integrity:** Git uses checksums (SHA-1) to ensure file contents are never corrupted or altered without detection.
- **Simple Explanation:** Instead of one central master copy, everyone has a perfect photocopy of the entire history.

5. Why is Git better than storing code in folders or drives ?

Answer: Git provides three key advantages over folder backups:

1. **Time Travel:** You can revert the project to any previous state instantly. Folders only show the current state.
2. **Branching:** You can work on multiple features simultaneously in parallel 'branches' without breaking the main code.
3. **Merging:** Git automatically combines work from different people, whereas merging folder copies manually is error-prone and tedious.
- **Real-World Example:** In a folder system, if you want to try a risky new feature, you have to copy the whole project folder (`Project-Copy`). In Git, you just type `git checkout -b new-feature`. It is faster, cleaner, and uses less space.

Part 2: Introduction to Git

6. What is Git ?

Answer: Git is a Distributed Version Control System (DVCS) created by Linus Torvalds. It allows multiple developers to work on the same project simultaneously. It tracks changes in source code, enabling teams to collaborate, revert to previous versions, and manage multiple versions of code (branches) efficiently.

- **Simple Explanation:** It is a "Save Game" system for code. You can save your progress (commit) at any point. If you mess up level 5, you can reload the save from level 4.
- **Real-World Example:** If you delete a critical 500-line function by mistake and save the file, Git lets you "undo" that delete instantly, even if you closed your editor days ago.

7. Why do we need Git ?

Answer: We need Git for three main reasons:

1. **Collaboration:** It merges code from different people automatically, preventing overwrite conflicts.
2. **History & Audit:** It keeps a log of ***who*** changed ***what*** and ***when***.
3. **Safety:** You can fix mistakes easily and experiment without breaking the main project.

- **Simple Explanation:** Without Git, working in a team is a chaos of "Final_v1", "Final_v2" zip files. With Git, everyone stays in sync automatically.

8. Difference between Git and GitHub

Answer: Git is the tool; GitHub is the service.

- **Git** is the software installed on your local computer to manage version control.
- **GitHub** is a cloud-based hosting platform where you upload (push) your Git repositories to share them with others.
- **Analogy:**
    - **Git** is like **Microsoft Word** (you use it on your computer to write).
    - **GitHub** is like **Google Drive** (you upload your document there to share it with the team).
- **Comparison Table:**

Feature	Git	GitHub
Type	Software (Tool)	Service (Website)
Location	Runs Locally	Runs on the Cloud (Web)
Usage	Managing history	Hosting & Collaboration
Competitors	Mercurial, SVN	GitLab, Bitbucket

9. What is a Repository (Repo) ?

Answer: A Repository (or 'repo') is a project folder that is tracked by Git. It contains all the project files and the entire revision history (stored in a hidden .git folder).

- **Local Repo:** The copy on your computer.
- **Remote Repo:** The copy hosted on a server (like GitHub).
- **Simple Explanation:** It's a "Super Folder". It looks like a normal folder, but it has a hidden camera (Git) recording every change you make inside it.
- **Command:** `git init` turns a regular folder into a repository.

10. What is Version Control ?

Answer: Version Control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It allows you to revert files to a previous state, compare changes over time, and see who modified the project.

- **Real-World Example:** Wikipedia's "View History" tab is a form of version control. You can see every edit made to an article and restore an old version if someone vandalizes it. Code version control works the same way.

Part 3: Git Basics – Everyday Workflow

11. `git init` vs `git clone`

Answer: These are the two ways to start a Git repository:

- `git init`**:** Converts an existing local directory into a new Git repository. It creates the hidden `.git` folder.
- `git clone`: Copies an existing remote repository (from GitHub/GitLab) to your local machine, including its entire history.
- **Simple Explanation:** `git init` is planting a new tree. `git clone` is taking a cutting from an existing tree to grow a copy elsewhere.
- **Commands:**
    - Start new: `git init` (Run this inside your project folder).
    - Copy existing: `git clone https://github.com/user/project.git`

12. `git add` and `git commit`

Answer: This is the core two-step process of saving changes:

1. `git add` **(Staging):** Selects which files you want to save. It moves changes to the 'Staging Area'.
2. `git commit` **(Saving):** Takes a snapshot of the files in the Staging Area and saves them permanently to the version history with a message.

- **Analogy:**
    - `git add`: putting items in your shopping cart.
    - `git commit`: checking out and paying (finalizing the purchase).
- **Commands:**
    - `git add .` (Stages all changes).
    - `git commit -m "Fixed login bug"` (Saves the snapshot).

13. `git status` and `git log`

Answer: These commands let you check the state of your project:

- `git status`: Shows the *current* state of your working directory (e.g., which files are modified, staged, or untracked).

- `git log`: Shows the *history* of commits (who changed what and when).
- **Real-World Example:**
    - Use `git status` before every commit to make sure you aren't accidentally committing a wrong file.
    - Use `git log` to find a specific commit ID (hash) if you need to revert changes.

14. `git diff`

Answer: `git diff` shows the exact line-by-line differences between files. It highlights what was added (green `+`) and what was removed (red `-`).

- **Simple Explanation:** `git status` tells you *which* file changed. `git diff` tells you *exactly what text* inside that file changed.
- **Real-World Scenario:** You fixed a bug but forgot exactly what you deleted. Run `git diff` to double-check your code before running `git add`.

15. `git reset` vs `git checkout` (Basic Usage)

Answer: Both are used to undo changes, but they work differently:

- `git checkout filename`: Discards local changes to a specific file and restores it to the last committed state. (Safe).
- `git reset`: Unstages files or undoes commits.
    - `git reset HEAD file`: Removes a file from the staging area (undoes `git add`).
- **Warning:** `git checkout` on a file is destructive to unsaved work—it wipes out your current edits to that file.
- **Command:**
    - "I messed up this file, revert it to the last save": `git checkout -- file.txt`
    - "I accidentally typed `git add`, undo it": `git reset HEAD file.txt`

Part 4: Branching & Collaboration

16. What is a Git branch ?

Answer: A Git branch is a separate line of work where you can make changes to code without affecting the main code.

- **Simple Explanation:** It's like making a copy of the project to work on a new feature or fix, while the original project stays safe.
- **Real-World Example:** Suppose you are adding Dark Mode to a website, you create a new branch and work on the Dark Mode feature there while the main branch remains stable and is used to fix urgent bugs or deploy updates; once the Dark Mode feature is completed and tested, you merge that branch back into the main branch.

17. How to create and switch branches (`git branch`, `git checkout -b`)

Answer: We use `git branch` to manage branches and `git checkout` (or `git switch`) to navigate between them.

- **Create:** `git branch new-feature`
- **Switch:** `git checkout new-feature`
- **Create & Switch (Shortcut):** `git checkout -b new-feature`
- **Tip:** In newer Git versions (2.23+), you can use `git switch` instead of `checkout` to avoid confusion.
    - `git switch -c new-feature` (Create and switch).

18. `git merge` vs `git rebase`

Answer: Both `git merge` and `git rebase` are used to bring changes from one branch into another, but they work differently:

- **Merge:** Combines two branches by creating a **new merge commit**. It keeps the full history of both branches.
- **Rebase:** Moves your branch commits on top of another branch. It **rewrites commit history** to make it look straight and clean.
- **Simple Explanation:**
    - **Merge:** Tying two ropes together with a knot. You can see where they joined.
    - **Rebase:** Cutting the second rope and gluing it to the end of the first rope to make one long, smooth rope.

- **Interview Tip:** "Never rebase a public branch (like `main`) that other people are working on. Only rebase your local private feature branch."

19. What is a merge conflict and how to resolve it ?

Answer: A merge conflict occurs when Git cannot automatically resolve differences in code between two commits—usually because the **same line** in the **same file** was modified differently in both branches.

To Resolve:

1. Git pauses the merge and marks the file as 'conflicted'.
2. Open the file and look for conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`).
3. Manually edit the code to choose the correct version (or combine them).
4. Run `git add` and `git commit` to finish the merge.

- **Real-World Scenario:** Developer A changes the login button color to **Blue**. Developer B changes it to **Red**. Git doesn't know which color is correct, so it asks you to decide.

20. `git pull` vs `git fetch`

Answer: The difference is how they handle the retrieved data:

- `git fetch`: Downloads new data (commits, files, refs) from the remote repository to your local machine but **does not** integrate it into your working files. It is safe and non-destructive.
- `git pull`: Automatically performs a `git fetch` followed immediately by a `git merge`. It updates your current work with the new changes.
- **Analogy:**
    - **Fetch:** Checking your mailbox and seeing you have letters, but leaving them closed on the table.
    - **Pull:** Opening the letters and reading them immediately.
- **Command:** `git pull origin main`

21. Remote branches (`origin/main`, `origin/feature`)

Answer: Remote branches are references (pointers) to the state of branches on your remote repository (like GitHub). They are prefixed with the remote name (usually

`origin`). You cannot check them out directly to work on them; you view them to see what the team has pushed.

- **Example:**
  - `main`: Your local branch.
  - `origin/main`: The branch on GitHub.
- **Scenario:** If you run `git fetch`, `origin/main` updates to show the latest work from your team. Your local `main` stays the same until you run `git merge origin/main` (or `git pull`).

Part 5: Git & Remote Repositories

22. How to add a remote (`git remote add`)

Answer: The `git remote add` command connects your local repository to a remote server (like GitHub or GitLab). You give it a name (conventionally `origin`) and the URL of the remote repository.

- **Simple Explanation:** It's like saving a phone number in your contacts. You save the long URL `https://github.com/...` under the short name "origin" so you don't have to type the URL every time.
- **Command:** `git remote add origin https://github.com/user/repo.git`
- **Check Remotes:** `git remote -v` (Shows the list of connected remotes).

23. `git push`

Answer: `git push` uploads your local repository content (commits) to a remote repository. It transfers commits from your local branch to the corresponding branch on the remote server.

- **Simple Explanation:** This is the "Upload" button. Until you push, your work exists only on your laptop. If your laptop breaks, the work is lost. Pushing backs it up to the cloud.

- **Command:** `git push -u origin main`
    - **Tip:** The `-u` flag links your local branch to the remote branch, so in the future, you can just type `git push` without arguments.

24. What is a Pull Request (PR) / Merge Request (MR) ?

Answer: A **Pull Request** (on GitHub) or **Merge Request** (on GitLab) is a feature that lets developers notify the team that they have completed a feature. It requests that their branch be merged into the main codebase. It is the primary place where **Code Reviews** happen.

- **Real-World Scenario:** You finish the "Dark Mode" feature. You don't just shove it into the main code. You open a PR. Your senior developer reviews the code, leaves comments ("Fix this typo", "Optimize this loop"), and once approved, they click "Merge".
- **Key Difference:** They are the same thing; just different names used by different platforms.

25. How to Fork a repository ?

Answer: A **Fork** creates a copy of a repository on your GitHub (or GitLab, etc.) account, letting you propose changes without write access to the original repo.

This is especially useful for contributing to open-source and team projects where you don't have direct write access to the main repository. You fork the repo, make changes in your fork, and then create a pull request to propose those changes to the original project.

- **Real-World Use Case:** You want to fix a bug in a popular Open Source project (like React or Linux). You cannot push directly to their repo (you don't have permission).
    1. **Fork** the repo to your account.
    2. Make changes in your copy.
    3. Send a **Pull Request** to the original owner asking them to accept your fix.

26. How to generate and use SSH keys with GitHub

Answer: Using SSH keys allows you to connect to GitHub without typing your username and password (or Personal Access Token) every time you push.

Steps:

1. **Generate:** Run `ssh-keygen -t ed25519 -C "email@example.com"` on your laptop.
2. **Copy:** Copy the public key (`cat ~/.ssh/id_ed25519.pub`).
3. **Add:** Go to GitHub Settings -> SSH and GPG Keys -> New SSH Key, and paste it.

- **Why do this?** It is more secure and convenient for automation (CI/CD) and daily work.

Part 6: Git Cleanup & Debugging

27. `git stash`

Answer: `git stash` temporarily saves your uncommitted changes so you can work on something else with a clean working directory. You can apply those changes back later.

- **Simple Explanation:** It's like putting your unfinished code into a drawer so your workspace becomes clean.
- **Real-World Scenario:** You are working on a feature when an urgent bug comes up. You use **git stash** to temporarily save your unfinished changes, switch to the main branch to fix and commit the bug, and then return to your feature branch and use **git stash pop** to get your saved work back and continue from where you left off.
- **Commands:**
    - `git stash`: Save changes.
    - `git stash pop`: Restore changes and delete the stash.
    - `git stash list`: See all saved stashes.

28. `git clean`

Answer: `git clean` is used to remove **untracked files** from your working directory. These are files that Git is not tracking (like build artifacts, temporary logs, or compilation files).

- **Warning:** This command deletes files permanently. There is no "Undo".

- **Real-World Scenario:** Your project folder is full of random `.log` and `.tmp` files generated by a test run. You want to reset the folder to exactly how it looks in the repo.
- **Commands:**
    - `git clean -n`: (Dry Run) Shows what *would* be deleted. **Always do this first.**
    - `git clean -fd`: (Force Directory) Actually deletes untracked files and folders.

29. `git reflog`

Answer: `git reflog` (Reference Log) tracks **every** movement of the HEAD pointer, including commits, resets, merges, and even checkout commands. It allows you to recover "lost" commits that are not visible in the standard `git log`.

- **Simple Explanation:** `git log` shows the history of the *project*. `git reflog` shows the history of *your actions*. It is Git's "Undo History" or "Time Machine" for mistakes.
- **Real-World Scenario:** You accidentally ran `git reset --hard` and deleted your latest work. It's gone from `git log`. But if you run `git reflog`, you will see an entry like `HEAD@{1}: reset: moving to HEAD~1`. You can grab the commit ID from there and recover your lost code.

30. `git cherry-pick`

Answer: `git cherry-pick` allows you to pick a specific commit from one branch and apply it to another branch, without merging the entire branch.

- **Simple Explanation:** Instead of merging the whole timeline, you just copy-paste one specific event.
- **Real-World Scenario:** You fixed a bug in the `feature-branch` (Commit A). The team needs that bug fix in `main` *right now*, but the rest of the feature branch isn't ready. You use `git cherry-pick <Commit-A-Hash>` to copy just that fix into `main`.

31. `git tag`

Answer: Tags are specific points in Git history that are marked as important, typically used for **releases** (e.g., `v1.0`, `v2.5`). Unlike branches, tags do not change; they stay fixed at a specific commit.

- **Real-World Scenario:** When you deploy your software to production, you tag that specific commit as `v1.0.0`. If you need to fix a bug in that specific version later, you can easily find exactly what code was running at that time.
- **Commands:**
    - `git tag v1.0`: Create a lightweight tag.
    - `git push origin v1.0`: Push the tag to GitHub (tags don't push automatically).

Part 7: Real-World Git Usage in DevOps

32. Git in CI/CD Pipelines

Answer: "Git acts as the primary **trigger** for almost all CI/CD pipelines. When a developer pushes code to a repository (e.g., GitHub or GitLab), the CI tool (like Jenkins or GitHub Actions) detects the change (webhook), automatically clones the repository, runs tests, builds the application, and deploys it."

- **Simple Explanation:** Pushing code to Git is like pressing the "Start" button on a factory assembly line. You drop raw materials (code) in, and the machine (CI/CD) automatically turns it into a product (app).
- **Real-World Scenario:** A developer pushes a bug fix to the `main` branch. GitHub Actions immediately sees the push, runs the unit tests, builds a Docker image, and deploys it to the staging server—all without human intervention.

33. Branching Strategies (Git Flow, Trunk-based)

Answer: "Branching strategies define the rules for how teams manage code changes.

- **Git Flow:** A strict structure with specific branches for `main`, `develop`, `feature`, `release`, and `hotfix`. Good for scheduled software releases (e.g., v1.0, v2.0).

- **Trunk-Based Development:** A modern approach where everyone merges into a single `main` (trunk) branch frequently (daily). It relies on 'Feature Flags' to hide unfinished work. Preferred for high-speed DevOps and CI/CD."
- **Interview Tip:** "For DevOps roles, mention that **Trunk-Based** is often preferred because it reduces 'merge hell' and encourages frequent integration."

34. Git for Infrastructure as Code (IaC)

Answer: "In DevOps, we treat infrastructure (servers, networks, firewalls) exactly like software application code. We write configuration files (Terraform, Ansible, Kubernetes YAML), store them in Git, and version control them. This practice is called **GitOps**."

- **Real-World Example:**
    - **Without Git:** A SysAdmin manually changes a firewall rule and breaks the site. No one knows what changed.
    - **With Git:** You change the firewall config in a Terraform file and commit it. If it breaks the site, you simply run `git revert` to undo the infrastructure change instantly.

35. How Git enables collaboration in DevOps

Answer: "Git creates a **Single Source of Truth**. It allows Devs and Ops to collaborate via **Pull Requests (PRs)**. Before any infrastructure or code change is applied, it goes through a PR where team members review the code, run automated tests, and discuss changes, ensuring quality and preventing conflicts."

- **Simple Explanation:** It prevents the "It works on my machine" problem by forcing everyone to merge their work into a shared, verified central history.

36. Common Git Mistakes Freshers Make

Answer: "The most common mistakes I need to avoid are:

1. **Committing Secrets:** Accidentally pushing AWS keys, passwords, or `.env` files to a public repo (Security Disaster).
2. **Working on Main:** Committing directly to the `main` branch instead of creating a feature branch.
3. **Committing Junk:** Pushing `node_modules`, build artifacts (`.exe`, `.jar`), or large media files (should be ignored via `.gitignore`).

4. **Force Pushing:** Using `git push --force` on a shared branch, which deletes other people's work."

Part 8: Scenario-Based Questions

37. You committed sensitive data (like a password) by mistake and pushed it to GitHub. What now ?

Answer: "This is a critical security incident.

1. **Immediate Action: Change the password/key** immediately. Once it is on GitHub, assume it is compromised.
2. **Cleanup:** Removing the file from the latest commit is not enough (it stays in history). You must rewrite history using tools like **BFG Repo Cleaner** (easier) or `git filter-branch` to purge the file from all commits.
3. **Force Push:** After cleaning, run `git push --force` to update the remote history."

- **Interview Tip:** "Always emphasize that **changing the password** is the first step. Cleaning the repo comes second."

38. Your teammate pushed new changes to main, but your local repo is outdated. What do you do before pushing your code ?

Answer: "I cannot push immediately because my history is behind.

1. **Fetch & Pull:** I will run `git pull origin main` to get my teammate's changes.
2. **Rebase (Optional):** I prefer `git pull --rebase origin main` to apply my changes *on top* of theirs for a cleaner history.
3. **Resolve Conflicts:** If we edited the same files, I will fix the conflicts manually, run `git add`, and continue.
4. **Push:** Finally, `git push origin feature-branch`."

39. You accidentally committed to the `main` branch instead of your feature branch. How do you fix it ?

Answer: "I would move those commits to a new branch without losing work.

1. **Create Branch:** `git branch feature-new` (Creates the branch pointing to my current commit).
2. **Reset Main:** `git reset --hard HEAD~1` (Moves `main` back one step to the correct state).
3. **Switch:** `git checkout feature-new`. Now my commit is safely on the feature branch, and main is clean."

- **Simple Explanation:** It's like pasting your paragraph on the wrong page. You copy it to the right page, then delete it from the wrong page.

40. You tried to push code, but Git says "rejected – non-fast-forward." What does it mean and how do you fix it ?

Answer: "This means the remote repository has commits that I do not have locally (someone else pushed while I was working). **Fix:**

1. Run `git pull --rebase origin main` to sync my local repo with the server.
2. Git will place my commits *after* the new ones.
3. Then run `git push origin main` again."

41. You want to test a feature without affecting the main repo. What Git feature helps ?

Answer: "I would create a **Feature Branch** (`git checkout -b test-feature`). This creates an isolated copy of the code where I can break things, delete files, or experiment safely. If the experiment fails, I just delete the branch (`git branch -D test-feature`) and `main` remains untouched."

42. You want to check who modified a specific line in a file last week.

Answer: "I would use the `git blame` command."

- **Command:** `git blame filename.txt`
- **Output:** It shows the Commit ID, Author Name, and Date for *every single line* in the file.
- **Real-World Scenario:** "Who hardcoded this API key on line 42?" -> Run `git blame config.js` -> "Ah, it was John on Tuesday."

43. You want to share a feature with your senior for review. What's the standard way ?

Answer: "I would push my feature branch to the remote repository and open a **Pull Request (PR)** (or Merge Request). In the PR, I would add a description of what I changed and tag my senior as a **Reviewer**. They can then see the 'Files Changed' diff and leave comments."

44. You deleted a branch accidentally, but you need it back. Can you recover it ?

Answer: "Yes, as long as it was recent. I would use `git reflog`.

1. Run `git reflog` to find the SHA (ID) of the last commit on that deleted branch.
2. Run `git checkout -b restored-branch <SHA>`."

- **Simple Explanation:** Git rarely deletes things immediately. It just hides them. `reflog` is the map to find hidden items.

45. You're joining a new team. How would you start working on a new repo with proper Git practices ?

Answer: "My workflow would be:

1. **Clone:** `git clone <repo-url>` to get the code.
2. **Branch:** Never work on main. Run `git checkout -b feature/my-task`.
3. **Commit:** Make small, frequent commits with clear messages (`git commit -m "Added login form"`).
4. **Pull:** Before pushing, run `git pull origin main` to ensure I'm up to date.
5. **Push & PR:** `git push origin feature/my-task` and create a Pull Request for review."