1. Git Basics

1. What is Git? Why is it used?

Git is a **distributed version control system (DVCS)** that tracks changes in a set of files, typically used for coordinating work among programmers collaboratively developing source code. It lets you **track changes**, **collaborate** on projects, and **revert** to previous versions of code. Its distributed nature means every developer has a **full copy** of the repository, enabling offline work and reducing dependency on a central server.

2. Difference between Git and GitHub?

Git is the **local** software that tracks changes, commits, and manages repositories on your machine. GitHub is a **cloud-based hosting service** for Git repositories, providing a web interface, collaboration tools like Pull Requests, and project management features. Think of Git as the engine and GitHub as the garage where you store and work on your car with others.

3. Difference between Git and other VCS (SVN, Mercurial)?

The main difference is that Git is **distributed**, while traditional VCS like SVN and CVS are **centralized**. In a centralized system, there's a single server holding the entire codebase, and developers must be online to interact with it. In Git, every developer has a **complete copy** of the repository, including its full history. This makes Git faster, more resilient, and allows for offline work.

4. What is a repository in Git?

A Git repository (or "repo") is a **storage location** for your project files, including all the versions of those files and their complete history. It contains a `.git` directory, which is a hidden folder that stores all the metadata and version history for the project.

5. What is the difference between local repo and remote repo?

A **local repository** is the copy of the project stored on your computer, where you make and track your changes. A **remote repository** is a version of the project hosted on a server, such as GitHub or GitLab, which serves as a central point for collaboration among team members.

6. What are Git commands you use daily?

Common commands include:

- `git status`: Shows the state of the working directory and staged area.
- `git add .`: Stages all changes in the current directory.
- `git commit -m "message"`: Saves changes to the local repository.
- `git push`: Sends committed changes to the remote repository.
- `git pull`: Fetches and merges changes from the remote repository.
- `git checkout -b branch-name`: Creates a new branch and switches to it.

7. What is a commit in Git?

A commit is a **snapshot** of your repository at a specific point in time. It's the primary way to save changes. Every commit has a **unique SHA-1 hash** for identification, a **commit message** describing the changes, and a **pointer** to its parent commit, forming a history chain.

8. What is a branch in Git? Why do we need branches?

A branch is a movable pointer to a commit. It represents an independent line of development.

We use branches to isolate new features or bug fixes from the main codebase. This allows multiple developers to work on different features simultaneously without interfering with each other's work.

9. What is the difference between `git clone` and `git fork`?

git clone is a command that downloads an existing remote repository onto your local machine. You typically use this when you have direct write access to the original repository.

git fork is a GitHub/GitLab action that creates a personal, independent copy of a repository on the server. You use this when you don't have direct write access to the original repo and want to propose changes via a Pull Request.

10. What is the difference between `git pull` and `git fetch`?

git fetch downloads new commits and objects from a remote repository but does not merge them into your local branch. It only updates the remote-tracking branches.

git pull is a combination of git fetch and git merge. It downloads the changes and automatically merges them into your current working branch. It's a quick way to update your local repository with the latest remote changes.

---

2. Git Staging & Commits

11. What is the difference between `git add`, `git commit`, and `git push`?

- `git add`: **Stages** changes from the working directory, preparing them to be included in the next commit.
- `git commit`: **Saves** the staged changes to the **local repository**, creating a new commit.
- `git push`: **Uploads** the committed changes from your local repository to the **remote repository**.

12. What is the difference between tracked, untracked, staged, and modified files?

- **Tracked files**: Files that were in the last snapshot (commit) or have been `git add`-ed.
- **Untracked files**: New files in your working directory that Git hasn't seen before.
- **Modified files**: Files that have been changed since the last commit.
- **Staged files**: Modified files that have been marked to be committed in the next snapshot (`git add`).

13. What is the difference between `git reset`, `git revert`, and `git checkout`?

- `git reset`: **Moves the HEAD** pointer, effectively rewinding history. It's often used to discard local changes and is considered a **destructive** operation as it rewrites history.
- `git revert`: Creates a **new commit** that **undoes** the changes of a previous commit. It's a non-destructive way to undo changes, preserving the project history.
- `git checkout`: Primarily used to **switch branches** or restore a file from a previous commit. It's used to navigate the project's history.

14. What is the difference between soft, mixed, and hard reset in Git?

All three are forms of `git reset`.

- `--soft`: **Moves HEAD** to the specified commit, but **keeps all changes staged**. The working directory remains untouched.
- `--mixed` (default): **Moves HEAD** and **unstages** the changes. The working directory is untouched, but the files become "modified."
- `--hard`: **Moves HEAD** and **discards all changes** in the working directory and staging area. **This is a destructive operation**; use it with caution.

15. What is a detached HEAD in Git?

A detached HEAD state occurs when you `git checkout` a commit that isn't the tip of any branch. This means you are on a specific commit, not a branch, so any new commits you make won't be part of a branch and are at risk of being lost unless you create a new branch.

16. How to undo the last commit in Git?

You can use `git reset --soft HEAD~1` to keep the changes in your staging area, or `git reset --mixed HEAD~1` to unstage them. If the commit has been pushed, you should use `git revert HEAD` to create a new commit that undoes the previous one, preserving history.

17. How to see commit history in Git?

The primary command is `git log`. Common options include:

- `git log --oneline`: Shows a concise, single-line log.
- `git log --graph --oneline --decorate`: Shows a graphical view of the branch history.
- `git log -p`: Shows the changes (diff) for each commit.

18. What is a Git SHA (commit hash)?

A SHA (Secure Hash Algorithm) is a **40-character hexadecimal string** that uniquely identifies a commit in Git. It's generated from the commit's content, including the tree, author, timestamp, and parent commit. This ensures the integrity of the project history.

19. How to amend the last commit in Git?

Use `git commit --amend`. This command **replaces the last commit** with a new one that includes the staged changes. You can also use it to change the commit message of the previous commit without adding new changes.

20. What is cherry-picking in Git?

Cherry-picking is the act of applying a single, specific commit from one branch to another. It's useful for taking a hotfix from a `hotfix` branch and applying it to the `main` branch without merging the entire `hotfix` branch. The command is `git cherry-pick <commit-hash>`.

---

3. Git Branching & Merging

21. What is branching in Git?

Branching is the process of creating a **separate line of development** in a Git repository. It allows developers to work on new features or bug fixes in isolation without affecting the main codebase.

22. What is merging in Git?

Merging is the process of **integrating changes** from one branch into another. This combines the history of the two branches, creating a new merge commit. It's how you bring a completed feature back into the `main` branch.

23. What is the difference between merge and rebase?

- **Merge**: Creates a **new merge commit** that combines the histories of two branches. It's a non-destructive operation that preserves history. The history can look "messy" with many merge commits.
- **Rebase**: **Rewrites history** by taking the commits from one branch and re-applying them on top of another. It creates a linear history, which can be "cleaner," but it's a destructive operation and should be used with caution, especially on shared branches.

24. What is a fast-forward merge vs 3-way merge?

- **Fast-forward merge**: Occurs when the branch you are merging from is a direct ancestor of the branch you are merging into. Git simply moves the pointer of the destination branch forward to the tip of the source branch. **No new merge commit is created**.
- **3-way merge**: Occurs when the branches have diverged. Git creates a **new merge commit** with two parents—the tips of both branches—to combine their changes.

25. What is the difference between `git merge --squash` and normal merge?

A normal merge combines commits from one branch into another, creating a new merge commit that retains the history of the source branch.

A --squash merge takes all the changes from the source branch and bundles them into a single new commit on the destination branch. This "squashes" the history of the source branch, keeping the commit log of the main branch clean.

26. What are Git conflicts? How do you resolve them?

A Git conflict occurs when Git cannot automatically merge changes from two branches because they have **modified the same lines** in the same file. To resolve them:

1. Git will mark the conflicting sections with `<<<<<<<`, `=======`, and `>>>>>>>`.
2. Manually edit the file to choose the desired code.
3. `git add` the file to stage the resolution.
4. `git commit` to finalize the merge.

27. What is the difference between `git stash` and branching?

- `git stash`: **Temporarily shelves** your uncommitted changes to a `stash` so you can switch branches or work on something else. It's for quick context-switching.
- **Branching**: Creates a **new, permanent line of development** to work on a feature.

28. How to rename a branch in Git?

1. To rename the **current local branch**: `git branch -m <new-name>`.
2. To rename a **local branch that isn't checked out**: `git branch -m <old-name> <new-name>`.
3. To rename a **remote branch**: Delete the old remote branch and push the new one.
    - `git push origin --delete <old-name>`
    - `git push origin -u <new-name>`

29. How to delete a local and remote branch?

- **Local**: `git branch -d <branch-name>`. Use `-D` to force delete if the branch is not fully merged.
- **Remote**: `git push origin --delete <branch-name>`.

30. What is a release branch, hotfix branch, feature branch (Git Flow)?

These are common branch types in the Git Flow workflow.

- **Feature branch**: Used to develop new features. They are branched from `develop` and merged back into `develop`.
- **Release branch**: Created from `develop` when it's ready for a new release. Only bug fixes are allowed. Once stable, it's merged into `main` and `develop`.
- **Hotfix branch**: Created directly from `main` to quickly fix a critical bug in a production release. It is merged into both `main` and `develop`.

---

4. GitHub & Collaboration

31. What is GitHub?

GitHub is a **web-based platform** that provides hosting for Git repositories and offers powerful collaboration features. It's often called the "social network for developers," as it allows them to share and collaborate on projects, track issues, and manage code releases.

32. Difference between GitHub, GitLab, and Bitbucket?

All three are web-based Git repository hosting services, but they differ in their features and focus.

- **GitHub**: Known for its large open-source community, simple interface, and social coding features.
- **GitLab**: Offers a **complete DevOps platform** with built-in CI/CD pipelines, container registry, and more, all in one application.
- **Bitbucket**: Often preferred by enterprise teams, it integrates tightly with other Atlassian products like Jira and Confluence.

33. What is a Pull Request (PR) in GitHub?

A Pull Request is a feature that allows a developer to **propose changes** and have them reviewed and discussed before being merged into a main branch. It's a key part of the collaboration workflow, enabling code review, automated checks, and discussions.

34. What is the difference between PR and Merge?

A **Pull Request** is the **request** to merge changes. It's a platform feature (like in GitHub) used for code review and collaboration. A **merge** is the **action** of combining changes from one branch into another, which can be done manually with the `git merge` command or by accepting a Pull Request.

35. What is GitHub Actions?

GitHub Actions is a **CI/CD (Continuous Integration/Continuous Deployment)** platform directly integrated into GitHub. It allows you to automate workflows, such as building, testing, and deploying code, triggered by events like a `git push` or `pull request`.

36. How to protect a branch in GitHub?

Branch protection rules can be set in the repository settings to prevent accidental or unauthorized changes. They can enforce requirements like:

- Requiring pull requests before merging.
- Requiring a minimum number of approving reviews.
- Requiring status checks to pass (like tests from CI/CD).
- Preventing force pushes.

37. What is CODEOWNERS in GitHub?

The `CODEOWNERS` file is a special file in a repository that **defines individuals or teams responsible** for specific parts of the codebase. When a Pull Request modifies files in a `CODEOWNERS` path, GitHub automatically requests a review from the designated code owners.

38. What is a GitHub issue and discussion?

- **GitHub Issues**: Used to **track bugs, feature requests, and tasks**. They are a structured way to manage work and link commits and pull requests to a specific problem.
- **GitHub Discussions**: A more recent feature for general **community communication**, like asking questions, sharing ideas, and providing announcements, separate from the more actionable nature of issues.

39. How to enforce code review in GitHub?

Code reviews can be enforced using **branch protection rules**. By requiring a certain number of approving reviews before a Pull Request can be merged, you ensure that every change is seen and approved by at least one other team member.

40. What is GitHub fork vs clone vs mirror?

- **Fork**: Creates a **server-side copy** of a repository under your account. You can make changes to your fork and submit them back to the original repository via a Pull Request.
- **Clone**: Creates a **local copy** of a repository on your machine.
- **Mirror**: Creates an **exact copy** of a repository, including all branches and tags. It's used for synchronization between two repositories.

---

5. Git Stash, Tags & Submodules

41. What is `git stash`? When do you use it?

`git stash` **saves your uncommitted changes** (both staged and unstaged) in a temporary shelf, allowing you to return to a clean working directory. You use it when you need to switch branches to work on an urgent task (like a hotfix) but don't want to commit your current, unfinished work.

42. Difference between `git stash pop` and `git stash apply`?

- `git stash pop`: **Applies** the stashed changes to your working directory and then **removes** the stash from the list.
- `git stash apply`: **Applies** the stashed changes to your working directory but **keeps** the stash in the list.

43. What are Git tags? Difference between lightweight and annotated tags?

Git tags are **pointers to specific commits** in the repository history, typically used to mark important points like release versions (e.g., `v1.0.0`).

- **Lightweight tags**: Just a **pointer** to a commit, like a temporary branch. They are local and not shared by default.
- **Annotated tags**: A full object in the Git database. They contain a message, a tagger name, date, and email. They are meant to be permanent, public markers and are recommended for marking releases.

44. How to delete a tag locally and remotely?

- **Local**: `git tag -d <tag-name>`
- **Remote**: `git push origin --delete <tag-name>`

45. What are Git submodules? Why are they used?

Git submodules allow you to **embed a Git repository inside another Git repository**. They are used when you have a core project that depends on external libraries or components that are managed in separate repositories. This helps to keep the main project's history clean and isolated.

46. Difference between Git submodule and subtree?

- **Submodule**: A separate, nested repository. The parent repository only stores a **pointer** to a specific commit of the submodule. It keeps the histories separate.
- **Subtree**: Merges a sub-repository into a subdirectory of the main repository. The histories are **intertwined**, making it easier to manage but harder to track changes from the original source.

47. How to update a Git submodule?

To update a submodule to the latest commit on its remote branch:

1. Navigate into the submodule's directory.
2. `git checkout <branch-name>`
3. `git pull`
4. Navigate back to the parent directory and `git add <submodule-path>` and `git commit` to record the change.

48. How to track submodule changes?

The parent repository tracks the **specific commit hash** of the submodule it points to. When you `git status` in the parent repo, it will show the submodule as "modified" if its HEAD is different from the commit hash stored in the parent repo. You must explicitly commit this change in the parent repository.

49. What are Git hooks? Examples?

Git hooks are **scripts** that Git runs automatically at specific points in the workflow, such as before a commit (`pre-commit`) or after a merge (`post-merge`).

- **`pre-commit`**: Used to run linting or code formatting checks before a commit is created.

- **post-receive**: A server-side hook often used to trigger a CI/CD build after a push.

50. How do you use Git hooks in CI/CD pipelines?

Git hooks are a great way to trigger actions in CI/CD. For example, a post-receive hook on a remote repository can be configured to:

1. Receive a push.
2. Trigger a build on a CI server (like Jenkins).
3. Notify a team channel that a new build has started.

---

6. Git Advanced Topics

51. What is rebasing in Git? When should you use it?

Rebasing is the process of moving or combining a sequence of commits to a new base commit. You use it to create a **clean, linear history** by moving your feature branch commits on top of the latest main branch commits. You should only rebase on **local, unpushed branches** to avoid rewriting history for others.

52. Difference between interactive rebase (git rebase -i) and normal rebase?

- **Normal rebase**: Automatically reapplies all commits from the source branch onto the destination branch.
- **Interactive rebase (git rebase -i)**: Gives you a **full-screen editor** where you can reorder, squash, edit, or delete commits before they are re-applied. This is a powerful tool for cleaning up your commit history.

53. What is git bisect? How is it used for debugging?

git bisect is a powerful tool for finding the **commit that introduced a bug**. It uses a **binary search algorithm** to quickly narrow down the range of commits where the bug was introduced. You start by marking a "good" commit (where the bug didn't exist) and a "bad" commit (where it does). Git then automatically checks out commits in the middle, and you tell it if the bug is present or not, repeating the process until the culprit commit is found.

54. What is git reflog?

git reflog is a record of **every action** taken in your local repository. It tracks when your HEAD was updated, showing a history of all branch checkouts, merges, resets, and other operations. It's an invaluable tool for recovering lost commits or fixing mistakes, as you can use it to find the SHA of a lost commit and restore it.

55. What is git blame?

git blame <file-name> shows who made the last change to **each line** of a file. It displays the author, the commit SHA, and the date of the last modification for every line, which is useful for understanding code history and ownership.

56. What is `git clean`?

`git clean` is a command that **removes untracked files** from your working directory. It's useful for cleaning up build artifacts or temporary files that you don't want to commit.

- `git clean -n`: (dry run) Shows what will be deleted without actually deleting it.
- `git clean -f`: Deletes the untracked files.

57. What is `git fsck`?

`git fsck` (file system check) is a command that **verifies the integrity** of the Git repository. It checks the internal database for consistency and can find corrupted or missing objects.

58. What is `git gc` (garbage collection)?

`git gc` is a command that **cleans up unnecessary objects** and packs the repository database for better performance. It runs automatically in the background but can be invoked manually to optimize the repo.

59. What is `git worktree`?

`git worktree` allows you to have **multiple working directories** associated with the same repository. This is useful for working on two different branches simultaneously without switching back and forth or stashing changes. For example, you can work on a feature branch in one directory while also testing a hotfix in another.

60. What is a bare repository in Git?

A bare repository is a Git repository that **does not have a working directory**. It's only the contents of the `.git` folder. Bare repos are typically used for **remote repositories**, as they are not meant for direct editing and are perfect for hosting.

---

7. Git Workflows

61. What is Git Flow workflow?

Git Flow is a **strict branching model** that defines a long-lived `main` (production) branch and a `develop` branch. It also uses support branches for features, releases, and hotfixes. It's well-suited for projects with a planned release cycle and is known for its complexity but also its structured approach.

62. What is GitHub Flow?

GitHub Flow is a **lightweight, continuous delivery-oriented workflow**. It has only one long-lived branch: `main`. All development happens in short-lived feature branches, which are merged into `main` after a Pull Request is approved. Every merge to `main` is a potential release.

63. Difference between centralized workflow vs feature branching vs forking?

- **Centralized workflow**: All developers work on a single `main` branch. This is simple but risky, as one person's broken code can block everyone.
- **Feature branching**: Each new feature is developed on a separate branch. This allows for parallel development and code review, but requires team coordination.
- **Forking workflow**: Each developer creates their own server-side copy (fork) of the repository. They push changes to their fork and submit Pull Requests to the original repository. This is common in open-source projects.

64. What is trunk-based development in Git?

Trunk-based development is a workflow where all developers merge their small, frequent commits into a **single, shared `trunk` or `main` branch**. It's the opposite of feature branching and is often used with CI/CD and feature flags to ensure a constantly shippable product.

65. What is the difference between rebase workflow and merge workflow?

- **Rebase workflow**: Emphasizes a **clean, linear commit history**. Developers rebase their feature branch onto `main` before merging, removing "merge commits." This can be cleaner but requires careful handling.
- **Merge workflow**: Uses **merge commits** to combine changes, preserving the exact history of the feature branch. The history can look "messy" but accurately reflects the development path.

66. What are advantages/disadvantages of Git Flow in CI/CD?

Advantages:

- **Clear separation**: The `main` branch is always stable and ready for production, which is good for strict release cycles.
- **Structured**: The defined branching model provides clarity for large teams.

Disadvantages:

- **Complex**: The multiple long-lived branches can be difficult to manage.
- **Not ideal for CI/CD**: The long-lived nature of `develop` and `release` branches can lead to slower delivery cycles compared to workflows like Trunk-Based Development.

67. How to handle release management in Git?

Release management in Git typically involves:

1. Creating a **release branch** from the main development branch (`develop`).
2. Applying only critical bug fixes and finalizing release notes on this branch.
3. Once stable, **tagging** the branch with a version number (e.g., `v1.0.0`).
4. Merging the release branch into both the `main` (production) branch and the `develop` branch.

68. What is semantic versioning in Git tags?

Semantic versioning is a system that uses a three-part version number: **MAJOR.MINOR.PATCH**. It's commonly used with Git tags to communicate the nature of changes.

- **MAJOR**: Incremented for incompatible API changes.
- **MINOR**: Incremented for new functionality in a backward-compatible way.
- **PATCH**: Incremented for backward-compatible bug fixes.

69. How to maintain multiple environments (dev, staging, prod) in Git?

You can use different branches or tags for each environment.

- A `develop` branch for the development environment.
- A `staging` branch for the staging environment.
- The main branch with vX.Y.Z tags for the production environment.
  A CI/CD pipeline can be configured to deploy code to a specific environment whenever a commit is pushed to the corresponding branch or a new tag is created.

70. What is monorepo vs polyrepo in Git?

- **Monorepo**: A single repository containing the code for **all projects**. It simplifies dependency management but can be a challenge to manage at scale.
- **Polyrepo**: A separate repository for **each project**. This provides clear boundaries but can lead to complex dependency management.

---

8. Git & CI/CD Integration

71. How to integrate GitHub with Jenkins/GitLab CI?

Integration is typically done using webhooks.

- **Jenkins**: Install the GitHub plugin and configure it to listen for GitHub webhooks (e.g., `push` or `pull_request` events).
- **GitLab CI**: GitLab has built-in CI/CD. The `.gitlab-ci.yml` file defines the pipeline, which automatically runs on events like a `git push`.

72. What is GitHub Actions?

GitHub Actions is a **CI/CD platform** built directly into GitHub. It allows you to automate a wide range of tasks, from testing and building to deployment, directly within your repository using YAML-based workflows.

73. Difference between GitHub Actions and Jenkins?

- **GitHub Actions**: **Fully integrated** into GitHub, making it seamless for projects hosted there. It uses a serverless model, and workflows are defined in YAML files (`.github/workflows`).
- **Jenkins**: A **highly customizable, self-hosted** CI/CD server. It's more complex to set up but offers more control and a vast plugin ecosystem.

74. How to implement CI/CD pipeline using GitHub Actions?

1. Create a `.github/workflows` directory in your repository.
2. Create a YAML file (e.g., `ci.yml`) inside it.
3. Define the workflow using `name`, `on`, `jobs`, and `steps`. The `on` key specifies the trigger (e.g., `push`, `pull_request`).
4. The `jobs` section defines the tasks to run, and `steps` within a job define the individual commands.

75. How to trigger builds only on PRs in GitHub?

In your GitHub Actions workflow YAML file, you can specify the pull_request event in the on section.

Example:

```
YAML

on:
  pull_request:
    branches:
      - main

```

This will trigger the workflow whenever a pull request is opened or updated on the `main` branch.

76. How to set up GitHub Secrets for secure CI/CD?

GitHub Secrets are **encrypted environment variables** you can set at the repository or organization level. They are used to store sensitive information like API keys, access tokens, or passwords. They are automatically masked in the logs and can be accessed in your workflow files using the `secrets` context.

77. Difference between workflow, job, and step in GitHub Actions?

- **Workflow**: A complete automated process defined in a YAML file. It's the highest-level concept.
- **Job**: A set of `steps` that are executed on the same runner. A workflow can have multiple jobs that run sequentially or in parallel.
- **Step**: An individual task within a job, such as running a command, using an action, or checking out a repository.

78. What is matrix build in GitHub Actions?

A matrix build allows you to run a single job with **multiple configurations** (e.g., different operating systems, Node.js versions, or library dependencies). This is highly efficient for testing your code across various environments with minimal configuration.

79. What is self-hosted runner in GitHub Actions?

A self-hosted runner is a machine you **set up and manage yourself** to execute GitHub Actions jobs. This is useful for:

- Using a specific environment not available on GitHub's hosted runners.
- Running jobs on-premises.
- Having more control over security and performance.

80. How to cache dependencies in GitHub Actions?

You can use the `actions/cache` action to cache dependencies (e.g., `node_modules`, Maven artifacts) between workflow runs. This significantly speeds up the build process by preventing the need to download and install dependencies from scratch every time.

---

9. Git Security & Best Practices

81. How to remove sensitive data from Git history?

Removing sensitive data like passwords or API keys from Git history is a serious task that requires **rewriting the history**.

- **The BFG Repo-Cleaner**: A third-party tool that's faster and easier than `git filter-repo`.
- git filter-repo: The official, modern way to rewrite history.
  Both methods require a force push (git push --force) after the history is rewritten, which is why it's a very destructive and dangerous operation on shared repositories.

82. What is `.gitignore`?

`.gitignore` is a file where you list the **files or patterns** that you want Git to **ignore**. These are typically files that should not be committed to the repository, such as build artifacts, temporary files, logs, or sensitive configuration files.

83. Difference between `.gitignore` and `.gitkeep`?

- `.gitignore`: Tells Git which files to **ignore**. It's the standard way to exclude files.
- `.gitkeep`: A common convention, but **not a standard Git command**. Git does not track empty directories. To force Git to track an otherwise empty directory, you can create a `.gitkeep` file inside it.

84. How to prevent committing secrets (API keys, passwords) to Git?

- Use a **`.gitignore` file** to prevent a secret's file from being committed.
- Use **environment variables** to pass secrets to your application.
- Use a **credential store** or a secrets manager like HashiCorp Vault.
- Use a `pre-commit` hook to automatically check for secrets before a commit.

85. What is Git LFS (Large File Storage)?

Git LFS is a Git extension that **replaces large files with text pointers** inside your repository. The actual large file content is stored on a remote server. This keeps the size of

the Git repository small, which is crucial for projects with large binary files like images, audio, or videos.

86. What is Git signing (`git commit -S`)?

`git commit -S` cryptographically **signs a commit** with your GPG or SSH key. This provides a way to verify the authenticity of a commit and ensure that the changes were made by the person they claim to be.

87. Difference between SSH and HTTPS in Git?

- **HTTPS**: Uses your username and password or a Personal Access Token (PAT) for authentication. It's easy to set up but can be less convenient for frequent pushes.
- **SSH**: Uses an **SSH key pair** for authentication. Once configured, it's more secure and requires no password for each push.

88. How to enforce 2FA in GitHub?

Organization owners can enforce Two-Factor Authentication (2FA) for all members of their organization. When enabled, members who don't have 2FA enabled on their account will be locked out of the organization's repositories until they enable it.

89. How to roll back a compromised Git repository?

1. Stop all development and disable access to the repository.
2. Identify the compromised commit(s) using `git log`, `git blame`, or `git bisect`.
3. Use `git revert` to create new commits that undo the changes of the compromised commits, or use `git filter-repo` to permanently remove them.
4. Force push the changes to the remote repository.
5. Rotate all compromised credentials.

90. Best practices for Git commit messages?

A good commit message should be **short, descriptive, and consistent**.

1. **Subject line**: Keep it under 50 characters, use the imperative mood ("Fix bug" not "Fixed bug"), and capitalize the first letter.
2. **Body**: Use a blank line after the subject, and wrap the body at 72 characters. The body should explain the "what" and "why" of the change.

---

10. Git in Real-World

91. How do you handle Git conflicts in large teams?

- **Communicate often**: Let the team know which files you're working on.
- **Pull frequently**: Pull the latest changes from the remote repository to reduce the chance of conflicts.
- **Use smaller, more frequent commits**: Small commits are easier to merge and resolve conflicts.

- **Adopt a consistent workflow**: Use a workflow like GitHub Flow to prevent long-lived feature branches that are more likely to have large conflicts.

92. How do you recover a deleted branch?

You can recover a deleted branch using `git reflog`, which keeps a history of your `HEAD`.

1. Use `git reflog` to find the commit hash of the tip of the deleted branch. It will likely say "checkout: moving from branch to hash."
2. Use `git checkout -b <new-branch-name> <commit-hash>` to create a new branch from that commit.

93. How to migrate a repo from GitHub to GitLab/Bitbucket?

1. Create an empty repository on the new platform (e.g., GitLab).
2. From your local copy of the old repo, add the new remote URL: `git remote add new-origin <url>`.
3. Mirror the repository: `git push --mirror new-origin`.
4. Remove the old remote: `git remote remove origin`.
5. Rename the new remote to `origin`: `git remote rename new-origin origin`.

94. How do you squash commits in Git?

You can squash commits using an **interactive rebase**:

1. `git rebase -i HEAD~<number-of-commits>`
2. An editor will open, showing a list of commits.
3. Change the `pick` command to `squash` (or `s`) for the commits you want to merge into the one above it.
4. Save and close the file. Git will prompt you to write a new commit message for the squashed commit.

95. How do you rebase an old feature branch with the main branch?

1. Checkout your feature branch: `git checkout feature-branch`.
2. Fetch the latest changes from the remote: `git fetch origin`.
3. Start the rebase: `git rebase origin/main`.
4. Git will re-apply your feature branch's commits on top of `main`. You may need to resolve conflicts.
5. After the rebase is complete, force push your branch: `git push --force-with-lease origin feature-branch`.

96. How to do code reviews in GitHub?

Code reviews in GitHub are done through **Pull Requests**.

1. Open a Pull Request.
2. Request a review from a team member.
3. Reviewers can view the code changes, add comments, and approve or reject the PR.
4. Once approved, the PR can be merged into the main branch.

97. How to enforce coding standards with Git hooks?

You can use a `pre-commit` hook to run a linter or code formatter.

1. Create a file named `pre-commit` in the `.git/hooks` directory.
2. Add a script to this file that runs a linter (e.g., ESLint, Prettier).
3. If the script fails, the commit will be aborted.
   Note that Git hooks are local, so you should use a tool like Husky or a CI/CD pipeline to enforce standards for the entire team.

98. How to set up GitHub webhooks?

1. Go to your repository's settings on GitHub.
2. Click on "Webhooks" in the sidebar.
3. Click "Add webhook."
4. Enter the payload URL (the endpoint that will receive the webhook).
5. Choose the events you want to trigger the webhook (e.g., `push`, `pull_request`).

99. How to monitor GitHub activity (audit logs)?

GitHub provides audit logs for organizations and enterprise accounts. These logs record actions like:

- Repository creation and deletion.
- Changes to access permissions.
- Branch protection rule changes.
  This is a critical security feature for monitoring who is doing what in your organization.

100. How to scale Git for enterprise projects (monorepos, submodules)?

- **Monorepos**: Use tools like **Bazel** or **Nx** to manage dependencies and build processes.
- **Submodules**: Use `git submodule update --remote` to manage dependencies.
- **Server-side optimization**: Configure Git servers to use `git gc` and other optimizations.
- **Partial Clone and Sparse Checkout**: These features allow developers to download only a subset of the repository, reducing the size and improving performance for massive repositories.