GIT Introduction

What is GIT and why is it used?

Git is a distributed version control system that tracks changes in source code during software development. It allows multiple developers to collaborate on a project, keep track of code changes, and merge their work efficiently. Git is widely used for its speed, flexibility, and ability to handle both small and large-scale projects, making it a fundamental tool in modern software development.

Explain the concept of version control system and how GIT fits into it.

A version control system (VCS) tracks and manages changes to source code, enabling collaboration and providing a history of revisions. Git is a distributed VCS that allows multiple developers to work on a project simultaneously, maintaining a history of changes and facilitating collaboration through features like branching and merging.

What are the advantages of using GIT over other version control systems?

Git offers advantages such as distributed version control, speed, branching flexibility, and a robust merging mechanism. Its efficiency, performance, and widespread adoption make it a popular choice for managing source code in various software development projects.

What is the difference between GIT and GitHub?

Git is a distributed version control system used for tracking changes in source code during software development. GitHub, on the other hand, is a web-based platform that provides hosting for Git repositories and adds collaboration features such as issue tracking, pull requests, and project management. Git is the version control tool, while GitHub is a platform that uses Git for collaborative software development.

Explain the working of a GIT repository and the various states of files in GIT.

In a Git repository, the working of files involves three main states:

1. Working Directory:
   This is where you modify and create files.
   Changes made here are not yet tracked by Git.
1. Staging Area (Index):
   After modifying files in the working directory, you stage them using git add.
   The staging area is a snapshot of the changes you want to include in the next commit.
1. Repository (HEAD):
   After staging changes, you commit them using git commit.
   The committed changes are now stored in the Git repository, creating a new snapshot.

The special label "HEAD" points to the latest commit in the current branch.

What is a 'commit' in GIT and how is it useful?

A commit in Git is a snapshot of changes to files in a repository. It represents a point in the project's history. Commits are useful for tracking changes, providing a history of development, and enabling collaboration among developers. They create a versioned record of the project at different stages, making it easy to revert to previous states if needed.

What are the primary components of a GIT repository?

The primary components of a Git repository include:

1. Working Directory: Where files are edited and modified.
2. Staging Area (Index): A snapshot of changes to be included in the next commit.
3. Repository: A database that stores committed snapshots and project history.
4. Branches: Independent lines of development, allowing for parallel work.
5. HEAD: A pointer to the latest commit in the currently active branch.

Explain the difference between 'git fetch' and 'git pull'.

git fetch retrieves changes from the remote repository without modifying your working directory. It updates the remote branches, allowing you to review changes before merging.

git pull is used to fetch and automatically merge changes from the remote repository into your current branch. It combines git fetch and git merge in a single command.

In summary, git fetch only retrieves changes, while git pull fetches and merges them automatically.

What is GIT branching and how does it help in development workflows?

Git branching is the practice of creating independent lines of development, allowing multiple features or changes to be worked on simultaneously. Each branch represents a distinct set of code changes. Branching helps in development workflows by:

1. Isolation: Developers can work on features or fixes in isolation, preventing interference with the main codebase until changes are ready.
2. Parallel Development: Multiple branches enable parallel development of different features, bug fixes, or experiments concurrently.
3. Collaboration: Team members can collaborate more efficiently by working on separate branches and merging changes when they are complete and tested.
4. Risk Mitigation: Branches provide a way to experiment with new ideas or features without affecting the stability of the main project until changes are deemed ready for integration.

In summary, Git branching facilitates a more organized and collaborative approach to software development.

Describe the steps involved in creating and merging a branch in GIT.

1. Create a Branch:
   Use git branch <branch_name> to create a new branch.

Switch to the new branch with git checkout <branch_name> or git switch <branch_name>.

1. Make Changes:
   Edit files in the working directory to implement new features or fixes.
1. Stage and Commit:
   Use git add to stage changes and git commit to save changes to the branch.
1. Switch to Main/Branch:
   Return to the main branch with git checkout main or git switch main.
1. Merge Branch:
   Execute git merge <branch_name> to integrate changes from the branch into the main branch.
1. Resolve Conflicts (if any):
   Address any conflicts that arise during the merge process.
1. Commit Merged Changes:
   After resolving conflicts, commit the changes to complete the merge.

These steps create and merge a branch, incorporating changes into the main branch while keeping the development history organized.

Explain the purpose of 'git rebase' and when to use it instead of 'git merge'.

git rebase is used to integrate changes from one branch into another by moving or combining a sequence of commits. Its purpose is to create a linear commit history, making it easier to understand and navigate. Unlike git merge, which creates a new commit for the merge, git rebase modifies the existing commit history.

Use git rebase instead of git merge when you want to:

1. Maintain a clean and linear commit history.
2. Avoid unnecessary merge commits.
3. Integrate changes from a feature branch into the main branch in a more streamlined manner.

However, avoid rebasing if the branch is shared with others, as it rewrites commit history and can cause conflicts for collaborators.

Explain the working of the 'git merge' command with an example.

The git merge command combines changes from different branches. Here's a brief explanation with an example:

Create a New Branch:

```
git checkout -b feature-branch
```

This command creates and switches to a new branch named "feature-branch."

Make Changes:

Make changes to files in the working directory and commit them.

Switch to Main/Branch:

```
git checkout main
```

Switch back to the main branch.

Merge the Branch:

```
git merge feature-branch
```

This command merges changes from "feature-branch" into the main branch.

Resolve Conflicts (if any):

If there are conflicts, resolve them, and complete the merge by committing the changes.

The git merge command creates a new commit that incorporates changes from the specified branch into the current branch (main in this example).

What is the significance of the '.gitignore' file in GIT?

The .gitignore file in Git specifies files and directories that should be ignored and not tracked by Git. It is significant for preventing unimportant or sensitive files, such as temporary files, compiled binaries, and configuration files, from being included in the version control system. This helps keep the repository clean, reduces unnecessary commits, and avoids sharing irrelevant or sensitive information with collaborators.

How does GIT handle merge conflicts and what strategies can be used to resolve them?

Git handles merge conflicts when concurrent changes are made to the same code. To resolve conflicts:

1. Git Marks Conflicts:
   Git marks conflicting areas in affected files.
1. Manually Resolve Conflicts:
   Edit the conflicted files to resolve differences.
1. Add Resolved Files:
   Use git add to mark the resolved files as staged.
1. Complete the Merge:
   Continue the merge process with git merge --continue.

Common conflict resolution strategies include:

Manual Resolution: Manually edit conflicting sections in the files.
Accept Incoming or Current Changes: Choose one version over the other.
Use a Merge Tool: Employ a visual merge tool to resolve conflicts.
Abort the Merge: Cancel the merge operation with git merge --abort.