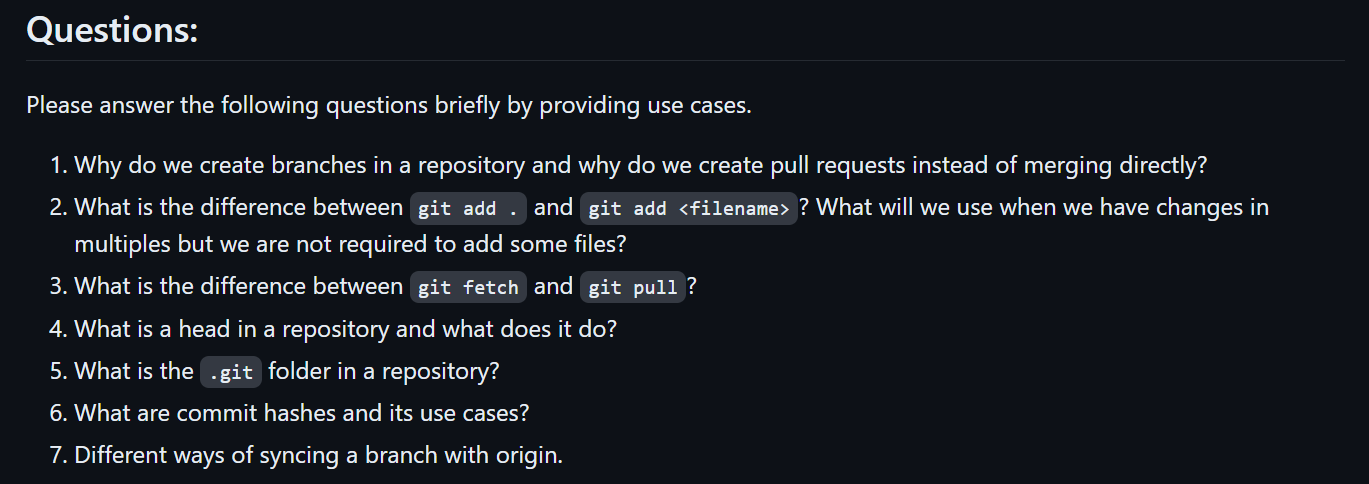
Nidhish Sharma

+91 7067027042

Nidhishsharma221094@acropolis.in

**ANSWERS**:

1. We create branches in repositories for the following reasons:

Feature Development:

* Developers create branches from the main codebase to work on new features. This practice ensures the main branch remains stable, allowing developers to implement and test new features without affecting the existing codebase.
* When a bug is detected in the production version of the software, a developer can branch off from the production code to address the issue. This method isolates the bug fix from other ongoing development activities, preventing potential conflicts.
* Developers can use branches to experiment with new technologies or methodologies. This allows them to try out innovative solutions without compromising the stability of the main project.

We Use Pull Requests Instead of Direct Merging for the following reasons:

* Pull requests allow developers to seek feedback from their peers before merging changes. This process helps identify bugs, ensure adherence to coding standards, and improve the overall quality of the code.
* Pull requests facilitate team collaboration. Team members can discuss specific changes, suggest improvements, and provide feedback directly on the code. This collaborative approach ensures a more refined implementation.
* Pull requests offer a clear record of what changes were made, the reasons behind them, and the contributors. This documentation is invaluable for future reference, debugging, and understanding the evolution of the project.

1. `git add .` stages all changes (new, modified, and deleted files) in the current directory and its subdirectories, making it useful for committing all project changes at once. In contrast, `git add <file>` stages only the specified file, providing precise control over which changes are included in the next commit, useful for selectively committing specific changes while leaving others uncommitted.

When you have changes in multiple files but do not want to stage all of them, you can use `git add <file1> <file2>` to selectively stage specific files. Alternatively, you can use `git add -p` for interactive staging, which allows you to review each change hunk-by-hunk and choose whether to stage, skip, or split the changes, providing granular control over what gets included in the next commit.

1. Git fetch :

* Retrieves the latest metadata (such as branch names, commit IDs, and tags) from the remote repository.
* Does not modify your working directory or local branches.
* Useful for reviewing changes before integrating them.
* Safe to run at any time.
* Example: ‘git fetch origin’

Git pull:

* Combines two steps: ‘git fetch’ followed by a merge.
* Fetches changes from the remote repository and updates your local branch.
* Also updates other remote-tracking branches.
* Use when you’re ready to incorporate new commits immediately.
* Example: ‘git pull origin main’

1. In Git, HEAD is a reference to the current checked-out commit in your repository. It acts as a pointer or symbolic reference to the latest commit in your branch. When you switch branches or check out a specific commit, HEAD adjusts to point to the relevant commit.
2. The .git folder is a critical component of any Git repository. It is a hidden directory located at the root of the repository that contains all the information necessary for Git to manage the version control of the project. This includes configuration settings, references to commits, and the actual data for the project history. Heres what lies inside it:

Sub-Directories:

* hooks/: Example scripts for Git hooks.
* info/: Excludes file for ignored patterns.
* objects/: Stores all “objects,” including blobs (files), trees (directories), and commits.
* refs/: Holds pointers to commit objects.

Files:

* HEAD: Represents the current branch.
* config: Contains configuration options.
* description: Provides a brief description of the repository.
* index: Acts as the staging area for changes.

1. A commit hash, also known as a commit ID, is a unique identifier generated by Git for each commit made to a repository. It is a 40-character string derived from the contents of the commit, including the author's details, the commit message, the timestamp, and the snapshot of the staged changes.

Use Cases:

* **Unique Identification**: Each commit hash uniquely identifies a specific commit, ensuring precise referencing and retrieval of changes.
* **Version Control**: Allows developers to check out specific versions of the codebase for reproducing bugs, reverting to previous states, or comparing versions.
* **Branching and Merging**: Tracks the history of branches and identifies common ancestors during merges for accurate integration of changes.
* **Collaboration and Code Review**: Used in pull requests and reviews to specify the exact changes being proposed or reviewed, ensuring clarity in discussions.
* **Audit and History Tracking**: Provides an immutable record of changes, crucial for tracking when and by whom changes were made, aiding in accountability and debugging.

1. Ways of syncing branch with origin:

* git pull: Fetch and merge changes from the remote branch.
* git fetch + git merge: Fetch changes and manually merge them.
* git fetch + git rebase: Fetch changes and rebase your commits on top.
* git pull --rebase: Fetch and rebase changes in one step.
* Force Update with git reset: Hard reset your branch to match the remote branch, discarding local changes.