Git and GitHub - A Brief Overview

Muskula Rahul

What is Git?

Git is a distributed version control system (DVCS) that helps developers track changes in their codebase over time. Created by Linus Torvalds in 2005, Git enables collaboration on software projects by allowing multiple contributors to work on the same code, track revisions, and merge changes efficiently. Git's key strength lies in its ability to manage large, branching code repositories while preserving the integrity and history of the code.

What is GitHub?

GitHub is a cloud-based platform that hosts Git repositories. While Git is the version control tool, GitHub adds a social, collaborative layer by providing a web-based interface for storing and managing Git repositories. GitHub offers features like pull requests, issue tracking, project management, and code reviews, making it a go-to tool for open-source projects and professional development teams.

Key Differences Between Git and GitHub

- **Tool vs. Platform**: Git is the actual version control tool, while GitHub is a web-based platform that leverages Git.
- **Local vs. Cloud**: Git operates locally on your machine, enabling version control and branching, whereas GitHub allows for hosting repositories remotely and facilitates team collaboration.
- **Functionality**: GitHub extends Git's functionality with additional features like a visual interface, project management tools, and cloud-based collaboration.
- **Hosting**: GitHub stores code remotely, allowing contributors to work from different locations, whereas Git manages code repositories locally unless configured to push to a remote server.

Basic Git Commands

Below are essential Git commands that every developer should know. These commands help in managing repositories, committing changes, and collaborating with teams.

1. `git init`
   Initializes a new Git repository in your project directory.
   ```
       git init
   ```
   This command creates a `.git` directory to start version control.
2. `git clone`
   Clones a repository from a remote location (e.g., GitHub) to your local machine.
   ```
       git clone <repository_url>
   ```

3. `git add`
   Stages files to be committed in the next commit.

   ```
   git add <filename>
   ```

   To add all files in the directory:

   ```
   git add .
   ```

4. `git commit`
   Commits staged changes to the local repository with a descriptive message.

   ```
   git commit -m "your commit message"
   ```

5. `git status`
   Displays the status of the working directory and staging area (tracked, untracked, modified files).

   ```
   git status
   ```

6. `git push`
   Pushes committed changes from the local repository to the remote repository.

   ```
   git push origin <branch>
   ```

7. `git pull`
   Fetches and merges changes from the remote repository to your local branch.

   ```
   git pull
   ```

8. `git branch`
   Lists, creates, or deletes branches.

   ```
   git branch
   ```

   To create a new branch:

   ```
   git branch <branch_name>
   ```

9. `git checkout`
   Switches to a different branch.

   ```
   git checkout <branch_name>
   ```

10. `git merge`
    Merges the specified branch into the current branch.

    ```
    git merge <branch_name>
    ```

11. `git log`
    Displays the commit history.

    ```
    git log
    ```

12. `git remote`
    Manages connections to remote repositories.

```
git remote -v
```

13. `git diff`
    Shows the changes between commits, branches, or the working directory.

```
git diff
```

14. `git reset`
    Unstages files from the index (staging area).

```
git reset <filename>
```

15. `git rm`
    Removes files from the working directory and the index.

```
git rm <filename>
```

16. `git mv`
    Renames or moves files.

```
git mv <old_filename> <new_filename>
```

17. `git stash`
    Temporarily saves changes that are not ready to commit.

```
git stash
```

18. `git tag`
    Creates a tag for marking specific points in the repository's history (e.g., version releases).

```
git tag <tag_name>
```

19. `git config`
    Configures Git settings, such as the user name and email.

```
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"
```

20. `git show`
    Displays information about a specific commit.

```
git show <commit_id>
```

Advanced Git Commands

These commands help you dive deeper into more complex scenarios like rebasing, patching, and working with specific histories or submodules.

1. `git rebase`
   Reapplies commits on top of another base tip to maintain a cleaner commit history.

```
    git rebase <branch_name>
```

2. `git cherry-pick`
   Applies a specific commit from one branch to another.

```
    git cherry-pick <commit_hash>
```

3. `git bisect`
   Finds the commit that introduced a bug by performing a binary search between commits.

```
    git bisect start
    git bisect bad
    git bisect good <commit_hash>
```

4. `git submodule`
   Manages repositories inside other repositories as submodules.

```
    git submodule add <repository_url>
```

5. `git reflog`
   Shows a log of changes to the local repository, including commits that are no longer visible in the commit history.

```
    git reflog
```

6. `git filter-branch`
   Rewrites branches by applying filters on the commit history (e.g., removing sensitive data).

```
    git filter-branch --force --index-filter 'git rm --cached --ignore
       -unmatch <filename>'
```

7. `git blame`
   Shows who made the last modification to each line in a file.

```
    git blame <filename>
```

8. `git archive`
   Creates an archive of the files in a particular commit or branch.

```
    git archive --format=zip HEAD > latest.zip
```

9. `git gc`
   Cleans up unnecessary files and optimizes the local repository.

```
    git gc
```

10. `git clean`
    Removes untracked files from the working directory.

```
    git clean -f
```

11. `git revert`
    Reverses the effects of a previous commit without modifying the history.

```
    git revert <commit_hash>
```

12. `git reset --hard`
    Resets the index and working directory to match a specific commit, discarding all changes.

```
    git reset --hard <commit_hash>
```

13. `git ls-tree`
    Lists the contents of a tree object, including file names, paths, and modes.

```
    git ls-tree <branch_name>
```

14. `git fsck`
    Verifies the integrity of the repository.

```
    git fsck
```

15. `git describe`
    Provides a human-readable name for a commit based on tags and the number of commits since the last tag.

```
    git describe
```

16. `git pull --rebase`
    Rebase instead of merging changes from the remote repository when pulling updates.

```
    git pull --rebase origin <branch>
```

17. `git diff --cached`
    Shows differences between the staging area and the last commit.

```
    git diff --cached
```

18. `git remote prune`
    Removes references to branches that no longer exist on the remote.

```
    git remote prune origin
```

19. `git fetch --all`
    Fetches updates from all remotes.

```
    git fetch --all
```

20. `git cherry`
    Shows which commits have not yet been applied to a branch.

```
    git cherry -v
```

Conclusion

Git is a powerful version control system that, when combined with GitHub, makes collaborative development smoother. Understanding the fundamental and advanced commands will help you manage codebases effectively, allowing for seamless collaboration and code integrity. Whether you're working on a solo project or part of a larger team, mastering Git commands is essential to becoming a proficient developer.