**1.What is Git and why is it used?**

**Git** is an open-source distributed version control system.

It is designed to handle minor to major projects with high speed and efficiency.

It is developed to co-ordinate the work among the developers.

The version control allows us to track and work together with our team members at the same workspace. Git stores the metadata and object database for your project.

It is used to:

* Manage projects with Repositories
* Clone a project to work on a local copy
* Control and track changes with Staging and Committing
* Branch and Merge to allow for work on different parts and versions of a project
* Pull the latest version of the project to a local copy
* Push local updates to the main project
* It eliminates everything from the time wasted passing commits over a network connection to the man hours required to integrate changes in a centralized version control system

**2.Explain the difference between Git pull and Git fetch.**

|  |  |
| --- | --- |
| **Git Pull** | **Git Fetch** |
| git pull copies changes from a remote repository directly into your working directory | git fetch does not copies changes from a remote repository directly into your working directory |
| git pull command does both. | git fetch command only copies changes into your local Git repo |
| The working directory is updated directly | Repository data is updated in the .git directory |
| Updates the changes to the local repository immediately. | Review of commits and changes can be done |
| Git Pull basically brings the local branch up-to-date with the remote copy that will also updates the other remote tracking branches. | Git fetch basically imports the commits to local branches so as to keep up-to-date that what everybody is working on |
| Command for Git Pull is git pull<remote><branch> | Command for Git fetch is git fetch<remote> |
| Merge conflicts are possible if the remote and the local repositories have done changes at the same place. | No possibility of merge conflicts. |

**3.How do you revert a commit in Git?**

Git revert is a safe, forward-moving alternative to Git reset when faced with the possibility of losing work. Git revert removes all the changes that a single commit made to the source code repository

To revert a commit in Git:

* Use the Git log or reflog command to find the ID of the commit you want to revert
* Enter the Git revert command (see below), including the commit ID you want to work on
* Provide an informative Git commit message to explain why you needed to perform the Git revert

**Git Revert Command**

**syntax to revert a Git commit called 31416p54**:

'git revert 31416p54'

**4.Describe the Git staging area.**

Git staging area is the middle ground between what you have done to your files (also known as the working directory) and what you had last committed (the HEAD commit). As the name implies, the staging area gives you space to prepare (stage) the changes that will be reflected on the next commit.

Assume you’re working on two files, but only one is ready to commit. You don’t want to be forced to commit both files, but only the one that is ready. This is where Git’s staging area comes in handy. We place files in a staging area before committing what has been staged. Even the deletion of a file must be recorded in Git’s history, therefore deleted files must be staged before being committed.

**Git Commands for the staging area:**

**git add:**

* It is used to stage any change in Git
* **git add** command adds a modification to the staging area from the working directory.
* **Syntax**:
* git add <file> - specify a <file> from which all changes will be staged.
* git add <directory> - specify a <directory> for the next commit
* git add . - use a . to add all the changes from the present directory

**git status:**

* It is used to check the status of the files (untracked, modified, or deleted) in the present branch.

**Syntax:**

git status

**git reset:**

* If we have accidentally staged a file or directory and want to undo it or unstage it, then we can use git reset command
* **Syntax:**

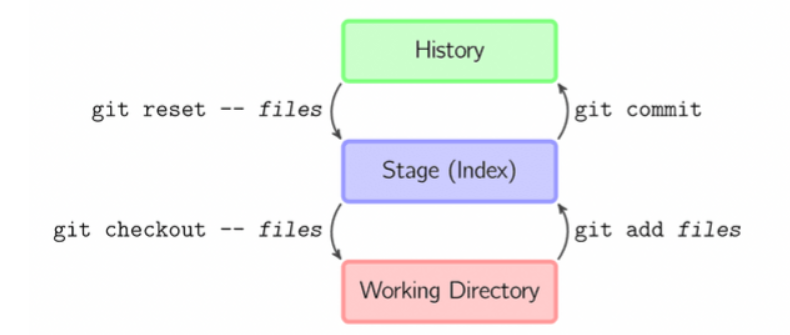
git reset HEAD example.html

git rm

**git commit:**

* It saves a snapshot of the current staged changes in the project. Committed snapshots are “secure” versions of a project that Git will never alter unless you specifically ask it to.
* Commits are produced with the git commit command to record the current status of a project.
* Git Snapshots are never committed to the remote repository. As the staging area serves as a wall between the working directory and the project history, each developer’s local repository serves as a wall between their contributions and the central repository.
* **Syntax:**

git commit -m "commit message"

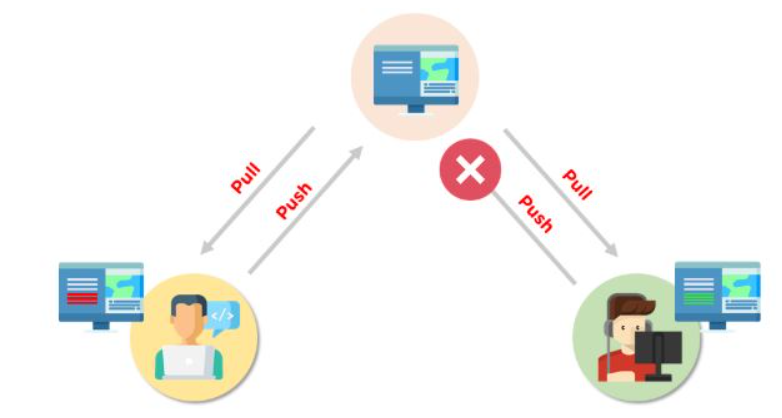


**5.What** **is a merge conflict, and how can it be resolved?**

Merge conflicts happen when people make different changes to the same line of the same file, or when one person edits a file and another person deletes the same file.

It takes place when Git is unable to automatically resolve differences in code between two commits. Git can merge the changes automatically only if the commits are on different lines or branches.

**Example of merge conflict:**

****

Let’s assume there are two developers: Developer A and Developer B. Both of them pull the same code file from the remote repository and try to make various amendments in that file. After making the changes, Developer A pushes the file back to the remote repository from his local repository. Now, when Developer B tries to push that file after making the changes from his end, he is unable to do so, as the file has already been changed in the remote repository.

**Resolve merge conflict:**

* The easiest way to resolve a conflicted file is to open it and make any necessary changes.
* After editing the file, we can use the git add a command to stage the new merged content.
* The final step is to create a new commit with the help of the git commit command.
* Git will create a new merge commit to finalize the merge.

**Commands to Resolve merge conflict:**

**git log --merge**

The git log --merge command helps to produce the list of commits that are causing the conflict.

**git diff**

The git diff command helps to identify the differences between the states repositories or files.

**git checkout**

The git checkout command is used to undo the changes made to the file, or for changing branches.

**git reset --mixed**

The git reset --mixed command is used to undo changes to the working directory and staging area.

**git merge –abort**

The git merge --abort command helps in exiting the merge process and returning back to the state before the merging began.

**git reset**

The git reset command is used at the time of merge conflict to reset the conflicted files to their original state.

**6.How does Git branching contribute to collaboration?**

* **Create a Repository**
* Set up a repository on GitHub to host your project.
* **Add Files to Your Project**
* Initialize a Git repository locally: `$ git init`
* Add files to the staging area: `$ git add <filename>`
* Commit changes: `$ git commit -m "initial commit"`
* Connect to a remote repository: `$ git remote add origin <remote repository URL>`
* Push changes to the remote repository: `$ git push -f origin master`
* **Adding Collaborators**
* In GitHub settings, navigate to "Manage access."
* Invite collaborators by providing necessary details.

**Important Git Commands and Terminologies**

* Git Tagging
* Create a tag: `$ git tag <tag\_name>`
* Create an annotated tag with a message: `$ git tag -a <version> -m "Release message"`
* Sign a tag: `$ git tag -s <tag\_name>`
* Verify a signed tag: `$ git tag -v <tag\_name>`
* **Git Checkout**
* Switch between branches: `$ git checkout <branch\_name>`

**Collaboration in GitHub**

* Registry: Collection of repositories containing files pushed by developers.
* Repository: Collection of files and folders located remotely in the registry.
* Use "git pull" to fetch code changes from GitHub.
* Use "git push" to push code changes to the GitHub remote repository.
* GitHub provides collaboration tools like inline commenting, notifications, integration, and project boards for effective coordination among team members.

**7. What is the purpose of Git rebase?**

**Git rebase is used to:**

* Maintain a clean, linear commit history.
* Squash commits for clarity.
* Edit commit messages or content.
* Avoid unnecessary merge commits.
* Integrate changes from the main branch.
* Resolve conflicts early during the rebase process.
* Facilitate interactive rebasing for flexible commit manipulation.

It helps create a more organized and readable history, but caution is advised when rebasing shared branches to avoid disrupting collaborators.

**8. Explain the difference between Git clone and Git fork.**

* **Git Clone**
* Purpose: Copy an existing Git repository to your local machine.
* Usage: ‘git clone <repository\_url>`
* Outcome: Full local copy of the repository with history and branches.
* Scenario: Used for working on existing projects or collaborating with others.
* **Git Fork**
* Purpose: Create a personal copy of a repository on Git hosting platforms (e.g., GitHub).
* Usage: Done on the web interface of the hosting platform.
* Outcome: Independent copy under your account; enables contributions without write access.
* Scenario: Common when contributing to open-source projects, allowing you to propose changes via pull requests.

**9.How do you delete a branch in Git?**

**To delete a branch in Git, use the command:**

git branch -d <branch\_name>

**If the branch has unmerged changes, Git will prevent deletion, use the `-D` option to force deletion:**

git branch -D <branch\_name>

Replace `<branch\_name>` with the name of the branch you want to delete.

The `-d` option stands for "delete" and is used for safe deletion. It ensures that the branch has been fully merged into the current branch before deletion.

The `-D` option, on the other hand, stands for "force delete" and is used when you want to delete the branch irrespective of its merge status. Use it with caution, as you may lose unmerged changes.

**10. What is a Git hook, and how can it be used?**

A Git hook is a script that Git executes before or after certain events, such as committing, pushing, or receiving changes. Git hooks allow developers to customize and automate parts of the Git workflow. They are scripts that run at specific points in the Git lifecycle and provide an opportunity to perform actions or checks.

Here are some common types of Git hooks:

**Pre-Commit Hook:**

* Executes before a commit is created.
* Useful for running code style checks, linting, or running automated tests before committing changes.

**Pre-Receive Hook:**

* Runs on the server before it processes a push.
* Useful for enforcing policies, such as rejecting pushes that don't meet certain criteria.

**Post-Receive Hook:**

* Runs on the server after a push has been processed.
* Useful for triggering deployment scripts, notifications, or other post-push actions.

**Pre-Push Hook**:

* Executes on the client before a push is executed.
* Can be used to run additional checks or tests before pushing changes.

**Post-Merge Hook:**

* Runs after a successful merge operation.
* Usefl for performing additional tasks after merging branches.

**How to Use Git Hooks:**

* Git hooks are stored in the `.git/hooks` directory of your Git repository.
* Each hook script is a plain executable file (e.g., a shell script).
* To enable a hook, you need to make it executable by running `chmod +x .git/hooks/<hook\_name>`.
* Git automatically calls the appropriate hooks at the specified events.

For example, to use a pre-commit hook for running tests before each commit:

1. Create a file named `.git/hooks/pre-commit`.
2. Add your testing commands to the script.
3. Make the script executable with `chmod +x .git/hooks/pre-commit`.

Now, the script will run automatically before each commit.