**What is Git?**

**Git** is a version control system. Think of it as a tool that helps you keep track of changes to your code over time. Here’s how it works:

* **Version Control**: Imagine you’re writing a book. Every time you make changes, you save a new version. Git does this for your code. It saves snapshots of your project at different points in time.
* **Collaboration**: If you’re working with a team, Git helps everyone keep track of changes. It allows multiple people to work on the same project without overwriting each other’s work.
* **Branching and Merging**: You can create branches to work on new features or fixes without affecting the main project. Once your work is ready, you can merge it back into the main branch.

**What is GitHub? (25MB is the max limit to upload file)**

**GitHub** is a platform built on top of Git. It provides a web-based interface to use Git and adds several features to make collaboration easier. Here’s what GitHub offers:

* **Repositories**: A repository (or repo) is like a project folder. It contains all your project files and the history of changes made to those files.
* **Remote Storage**: GitHub stores your repositories in the cloud, so you can access them from anywhere and share them with others.
* **Collaboration Tools**: GitHub makes it easy to collaborate with others. You can review code, discuss changes, and manage projects using features like pull requests and issues.

**Example to Illustrate Git and GitHub**

Imagine you and your friend are working on a website together. Here’s how you would use Git and GitHub:

1. **Create a Repository**: You create a new repository on GitHub for your website project.
2. **Clone the Repository**: You and your friend clone (download) the repository to your local machines using Git.
3. **Make Changes**: You create a new branch to add a new feature, like a contact form. Your friend creates another branch to improve the homepage design. You both make changes and commit (save) them to your respective branches.
4. **Push Changes**: You push (upload) your changes to the GitHub repository.
5. **Pull Requests**: Once your feature is ready, you create a pull request on GitHub. This is a request to merge your changes into the main branch. Your friend reviews your changes, and if everything looks good, they approve the pull request.
6. **Merge Changes**: Your changes are merged into the main branch, and the updated project is now available to everyone.
7. **Collaborate and Repeat**: You continue to work on new features, fix bugs, and collaborate using Git and GitHub.

**Configuring Git**

* **Install Git**: Download and install Git from the official website.
* **Set Up Identity**: Configure your user name and email.
* **Check Configuration**: Verify your settings with git config --list.
* **Set Default Editor**: Choose your preferred text editor.
* **Handle Line Endings**: Configure Git to manage line endings.
* **Create Aliases**: Set up shortcuts for common commands.

**Git Workflow (Udemy)**

Sure, let’s break down the entire Git workflow step-by-step, starting from creating a repository on GitHub to synchronizing changes between the local and remote repositories. I’ll explain each command and its purpose along the way.

**1. Creating a Repository on GitHub**

1. **Open GitHub**:
2. **Create a New Repository**:
   * Fill out the form:
     + **Repository name**: Choose a short, descriptive name (e.g., github-demo).
     + **Description**: Optionally, add a description (e.g., “A simple demo repository to show the basic Git workflow”).
     + **Public/Private**: Choose whether the repository is public or private.
     + **Initialize this repository with a README**: Check this box to start with a README file.
   * Click **“Create repository”**.

**2. Setting Up Your Local System**

1. **Open Terminal**: On Windows, use **Git Bash**.
2. **Navigate to Your Home Directory**: By default, the terminal opens in your home directory.
3. **Create a Projects Directory**: To keep your projects organized**, create a directory named projects**: mkdir projects

**Navigate into the projects directory**: cd projects

**3. Cloning the Repository**

1. **Clone the Repository**: Go back to your GitHub repository page and find the **“Code”** button. Click it and copy the repository URL. In your terminal, clone the repository to your local system: git clone <repository-url>
2. **Navigate into the Repository Directory**: Change into the newly cloned repository directory: cd github-demo

**4. Making Changes and Committing**

1. **Make Changes**: Create or edit files in your repository. For example, create a new file: echo "Hello, GitHub!" > hello.txt
2. **Stage Changes**: Add the changes to the staging area: git add hello.txt
3. **Commit Changes**: Commit the changes with a message: git commit -m "Add hello.txt with a greeting"

**5. Pushing Changes to GitHub:** Push your local commits to the remote repository on GitHub: git push origin main

**6. Synchronizing Changes: Pull Changes**: To update your local repository with changes from GitHub, use: git pull origin main

**Background on Default Branch Name Change (Udemy)**

1. **Why the Change?**
   * The Git community decided to move away from using “master” as the default branch name for political and social reasons.
   * GitHub, a popular Git hosting service, changed the default branch name for new repositories to “main”.
2. **Impact of the Change**
   * If you create a repository locally with “master” as the default branch and push it to GitHub, you might encounter errors because GitHub now uses “main” as the default branch for new repositories.

**Two Approaches to Handle the Change**

* **Reverting to “master”:** Change GitHub settings to use “master” as the default branch.
* **Using “main”:** Update your Git version, create repositories with “main” as the default branch, and adjust your workflow accordingly.

**git init (udemy)**

* + Imagine you’re starting a new project, like building a treehouse. Before you can add branches and leaves, you need to set up the treehouse structure.
  + Similarly, git init is like setting up the foundation for your project. It creates a special folder called .git inside your project folder. This folder keeps track of all the changes you make.
  + **Initialize a New Repository**: Create a new Git repository named “fresh-project”:

git init fresh-project

This command creates a new folder called “fresh-project” and initializes it as a Git repository.

* + **Example**: You’re creating a new website. Open your project folder in the terminal and type git init. Now your project is a Git repository, ready to track changes.

**Cloning a Repository**

* **Cloning** a repository means making a copy of an existing repository from a remote server (like GitHub) to your local machine. Here’s how you do it:

1. **Find the Repository URL**: Go to the repository you want to clone on GitHub and copy its URL. For example, https://github.com/username/repo.git.
2. **Clone the Repository**:
   * Open your terminal and run the following command:
   * git clone https://github.com/username/repo.git
   * This command creates a new directory named repo and copies all the files and history from the remote repository to your local machine.

* **Checking the Status:** The **status** command shows the current state of your working directory and staging area. It helps you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git.

1. **Navigate to Your Repository**: Change to the directory of your cloned repository:

cd repo

1. **Check the Status**: Run the following command to see the status:

git status

This command will show you:

**Untracked files**: Files that are not yet tracked by Git.

**Changes not staged for commit**: Files that have been modified but not yet staged.

**Changes to be committed**: Files that have been staged and are ready to be committed.

**Git Add**

The git add command is used to stage changes. Staging means preparing your changes to be committed. Think of it as adding items to a shopping cart before you check out.

1. **Stage a Single File**: If you modified a file called index.html, you can stage it with:

git add index.html

1. **Stage All Changes**: To stage all changes in your working directory, use:

git add .

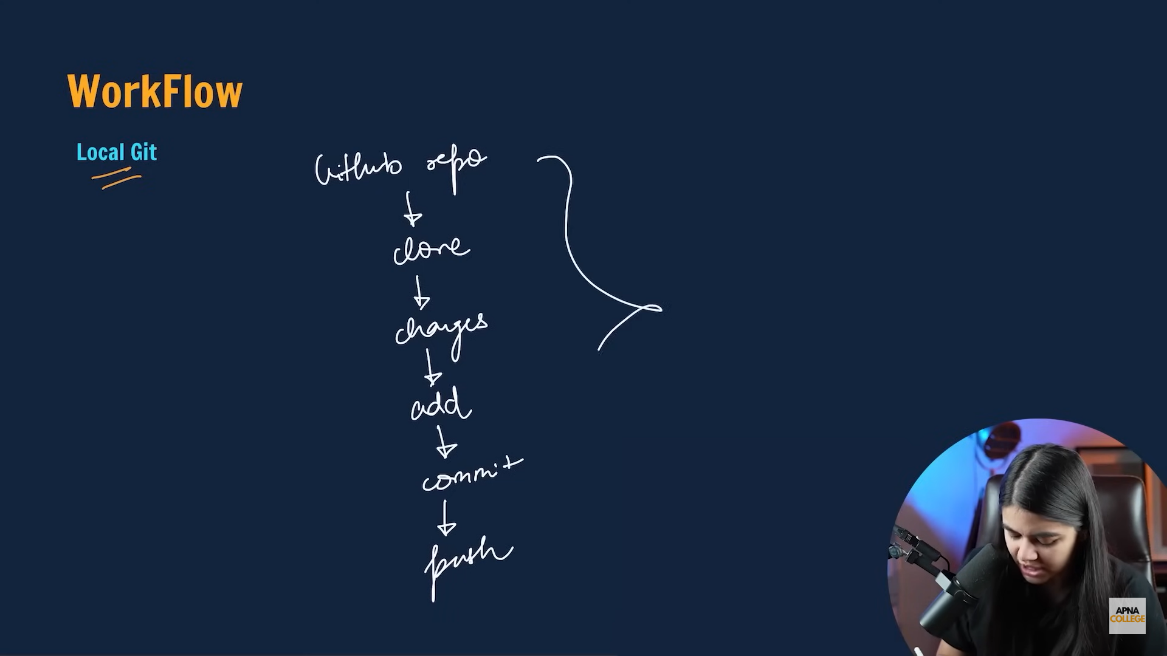
**Git Commit:** The git commit command is used to save your staged changes to the repository. This creates a snapshot of your project at that point in time.

1. **Commit Staged Changes**: After staging your changes, you can commit them with a message describing what you did:

git commit -m "Added a new feature to index.html"

**git push**

* + Imagine you’re working on a group project, and you’ve made some cool additions to the project files. Now you want to share those changes with your team.
  + git push is like saying, “Hey team, check out my awesome work!” It sends your local changes (commits) to a shared place (usually an online repository like GitHub).
  + **Example**: You’ve fixed some bugs in your code. First, use git add to stage your changes. Then, commit them with git commit -m "Fixed bugs". Finally, type git push to upload your changes to the remote repository (like GitHub). Now your team can see your fixes!



**What is a Git Branch?**

* + Imagine you’re working on a big project (like building a website). You want to try out new features or fix bugs without messing up the main project.
  + A **branch** in Git is like creating a separate workspace for your changes. It’s like having a parallel universe where you can experiment without affecting the main project.
  + By default, Git gives you a branch called main. This branch points to the latest version of your project.

1. **Creating a New Branch:**
   * Suppose you’re designing a new feature (let’s call it “hello-world-images”) for your website.
   * To create a new branch, use:

git branch hello-world-images

* + Now you have a fresh branch to work on!

1. **Switching to the New Branch:**
   * Use the checkout command to move to your new branch:

**git checkout hello-world-images**

* + Your workspace is now in the “hello-world-images” universe.

1. **Making Changes:**
   * Edit your files, add images, or write code in this branch.
   * For example, you add an image called img\_hello\_world.jpg to your project.
2. **Merging Back:**
   * When your feature is ready, you can **merge** your branch back into the main project.
   * This way, your changes become part of the main project.
   * Git helps you keep track of everything!

**Tracking Files (Udemy)**

**What Are Tracked Files?**

In Git, files in your working directory can be in one of two states: **tracked** or **untracked**.

* **Tracked Files**: These are files that Git knows about. They are being monitored for changes. Tracked files can be in three states:
  + **Unmodified**: The file hasn’t changed since the last commit.
  + **Modified**: The file has changed but hasn’t been staged for commit.
  + **Staged**: The file has been marked to be included in the next commit.
* **Untracked Files**: These are files that Git is not monitoring. They are new files that haven’t been added to the staging area.

**What is a Pull Request**

* + Imagine you’re working on a group project (like building a website) with your team. Everyone has their own branches where they make changes.
  + A **pull request (PR)** is like raising your hand and saying, “Hey, I’ve made some cool changes in my branch! Can we merge them into the main project?”
  + **It’s a way to propose changes from one branch (your feature branch) to another (usually the main branch).**

1. **How Does It Work?**
   * You create a pull request to show your changes to others. Collaborators can review, discuss, and give feedback on your proposed changes.
   * The PR displays the differences (or **diffs**) between your branch and the target branch (usually main or master).
   * Once everyone agrees, the changes are integrated into the main codebase.

**What is a Git Fork**

In Git, a **fork** refers to a personal copy of another user’s repository. When you fork a repository, you create an independent copy that exists within your own account or organization. [This copy includes all the files, commit history, and branches present in the original repository at the time of forking1](https://www.geeksforgeeks.org/git-fork/).

Here are some reasons why forking is commonly used:

1. **Testing Changes**: You have your own copy of the project on which you can test your own changes without affecting the original project. This helps the maintainer of the project review the changes you made and decide whether to accept, reject, or suggest modifications.
2. **Contributing to Open Source**: When you clone an open-source project, you don’t have the right to push code directly into the project. Forking allows you to work on your changes independently and then submit a pull request to the original project.

**Renaming, Moving and Deleting Files in Git**

1. **Rename a File**: Use the git mv command to rename a file:

git mv oldname.txt newname.txt

1. **Move a File**: Use the git mv command to move a file:

git mv filename.txt newdirectory/

1. **Delete a File**: Use the git rm command to delete a file:

git rm filename.txt

**Git History (Udemy)**

**git log:** The git log command is used to view the commit history of a repository. It provides various options to customize the output.

**Basic Usage**

1. **Default Log**: To see the commit history in reverse chronological order: **git log**
   * Output includes:
     + Commit SHA-1 hash
     + Author’s name and email
     + Date of the commit
     + Commit message

**Customizing the Log Output**

1. **Abbreviated Commit IDs**: To show shorter commit IDs: git log --abbrev-commit

Output: commit 1a2b3c4

Author: John Doe [john@example.com](mailto:john@example.com)

Date: Mon Oct 16 10:00:00 2023 +0000

1. **One-line Summary**: To show each commit on a single line: git log --oneline

Output:

1a2b3c4 Initial commit

5d6e7f8 Add README file

1. **Date-based Filtering**: To show commits made in the last 3 days:

git log --since="3 days ago"

1. **File-specific Log**: To show the commit history of a specific file: git log -- <file>

* **git show:** The git show command is used to display detailed information about a specific commit. **Basic Usage**

1. **Show a Specific Commit**: To display details of a specific commit: git show <commit-id>
   * Example: git show 1a2b3c4
   * Output includes:

Commit SHA-1 hash

Author’s name and email

Date of the commit

Commit message

Changes made (diff)

**Git Alias, Git Ignore, Git Cleanup**

Git aliases are shortcuts that allow you to create custom commands or abbreviations for frequently used Git commands. This can save you time and make your workflow more efficient. Let’s go through how to create and use Git aliases with some examples.

**Creating Git Aliases:** You can create Git aliases by modifying your Git configuration file (.gitconfig). There are two main ways to do this:

1. **Using the Command Line**: You can set aliases directly from the command line using the git config command.
2. **Editing the**.gitconfig**File**: You can manually edit the .gitconfig file to add aliases.

**Examples of Git Aliases**

1. **Shorten**git status**to**git st:
   * **Command:** git config --global alias.st status
   * **Usage**: git st

* **.gitignore:** The .gitignore file is used to tell Git which files or directories to ignore in a project. This is useful for excluding files that are not necessary to track, such as temporary files, build artifacts, or sensitive information.

1. **Create a**.gitignore**File**:In your project directory, create a file named .gitignore:

touch .gitignore

1. **Add Patterns to**.gitignore: Open the .gitignore file in a text editor and add patterns for files and directories you want to ignore. For example:

# Ignore all .log files

\*.log

# Ignore the node\_modules directory

node\_modules/

# Ignore all files in the temp directory

temp/

1. **Check Git Status**: Verify that the specified files and directories are ignored: git status

* **Git Cleanup:** Git cleanup involves removing unnecessary files and optimizing the repository to keep it clean and efficient. This can include removing untracked files, cleaning up branches, and pruning unused objects.

**Removing Untracked Files**

1. **List Untracked Files**: Use the git clean -n command to see which untracked files would be removed: git clean -n

**Output**:

Would remove error.log

Would remove node\_modules/

1. **Remove Untracked Files**: Use the git clean -f command to actually remove the untracked files: git clean -f
2. **Remove Untracked Directories**: Use the git clean -fd command to remove untracked directories as well: git clean -fd

**Cleaning Up Branches**

1. **List Local Branches**: List all local branches: git branch
2. **Delete a Local Branch**: Delete a local branch that is no longer needed: git branch -d branch-name

**P4Merge Tool**

**What is P4Merge?**

P4Merge is a visual tool developed by Perforce that helps you compare and merge files. It is particularly useful for developers and designers who need to see differences between file versions and resolve conflicts. P4Merge supports text files, code files, and even image files.

**Key Features of P4Merge**

1. **Three-Way Merging: P4Merge allows you to merge changes from two different versions of a file into a common base version. This is especially useful when multiple people are working on the same file.**
2. **Side-by-Side File Comparison**: You can compare two files side-by-side to see the differences between them. This helps in understanding what has changed between versions.
3. **Visualizing Merges**: P4Merge provides a visual representation of merges, making it easier to resolve conflicts and understand changes.
4. **Image Comparison**: P4Merge can compare image files by overlaying them or displaying them side-by-side. This is useful for designers who need to see pixel-level changes.

**Using P4Merge for File Comparison**

1. **Compare Two Files**: To compare two files, use the following command:

git difftool <file1> <file2>

* + Example: git difftool file1.txt file2.txt
  + This command opens P4Merge and displays the differences between file1.txt and file2.txt.

**Using P4Merge for Merging**

1. **Merge Conflicts**: When you have a merge conflict, you can use P4Merge to resolve it. First, initiate the merge:

git merge <branch-name>

If there are conflicts, use P4Merge to resolve them: **git mergetool**

P4Merge will open, showing the base version, your version, and the incoming version. You can manually resolve conflicts by choosing which changes to keep.

**Comparisons (Udemy)**

* **git diff:** The git diff command is used to show the differences between various states of your files. It can compare changes in your working directory, staging area, and commits.

**Basic Usage**

1. **Compare Working Directory with Staging Area**: To see changes in your working directory that are not yet staged: **git diff**
   * This command shows the differences between your working directory and the staging area.
2. **Compare Staging Area with Last Commit**: To see changes that are staged but not yet committed: **git diff --staged**
   * This command shows the differences between the staging area and the last commit.
3. **Compare Working Directory with Last Commit**: To see all changes in your working directory compared to the last commit: **git diff HEAD**
   * This command shows the differences between your working directory and the last commit.
4. **Compare Specific File**: To see changes in a specific file: git diff -- <file>

* **git difftool:** The git difftool command is used to launch an external tool to view differences. This is useful for visualizing changes in a more user-friendly way.

**Basic Usage**

1. **Compare Working Directory with Staging Area**: To see changes in your working directory that are not yet staged using a visual tool: **git difftool**
2. **Compare Staging Area with Last Commit**: To see changes that are staged but not yet committed using a visual tool: **git difftool –staged**
3. **Compare Working Directory with Last Commit**: To see all changes in your working directory compared to the last commit using a visual tool: **git difftool HEAD**
4. **Compare Specific File**: To see changes in a specific file using a visual tool: **git difftool -- <file>**

**Comparing Different States in Git (Udemy)**

**1. Comparing an Arbitrary Commit to the Last Commit**

1. **View Commit History**: Use the git log --oneline command to see a simplified history of commits. This will show a list of commits with abbreviated commit IDs.
2. **Compare Two Commits**: To compare an arbitrary commit to the last commit, use:

git diff <commit-id> HEAD

* + Example: git diff abc1234 HEAD

This command shows the differences between the specified commit (abc1234) and the latest commit (HEAD).

1. **Visual Comparison**: To see the differences visually using a difftool like P4Merge:

git difftool <commit-id> HEAD

* + Example: git difftool abc1234 HEAD

**2. Comparing Two Arbitrary Commits**

1. **Compare Two Specific Commits**: Use the git diff command with two commit IDs: git diff <commit-id1> <commit-id2>
   * Example: git diff abc1234 def5678

This command shows the differences between the two specified commits.

1. **Visual Comparison**: To see the differences visually: git difftool <commit-id1> <commit-id2>
   * Example: git difftool abc1234 def5678

**3. Comparing Two Branches**

1. **Compare Local and Remote Branches**: Use the git diff command to compare the local master branch with the remote origin/master branch: git diff master origin/master

This command shows the differences between the local master branch and the remote origin/master branch.

1. **Visual Comparison**: To see the differences visually: git difftool master origin/master

**Branches Basics (Udemy)**

Branches in Git allow you to work on different features or fixes independently from the main codebase. This helps keep your master branch clean and stable while you develop new features or experiment with changes.

* **List all branches**: git branch -a
* **Create a new branch**: git branch <branch-name>
* **Switch to a branch**: git checkout <branch-name>
* **Rename a branch**: git branch -m <old-name> <new-name>
* **Delete a branch**: git branch -d <branch-name>
* **Merge a branch**: git merge <branch-name>

**Branching with Fast forward merges**

* **Create and switch to a new branch**: git checkout -b <branch-name>
* **Check current branch**: git branch
* **Edit a file**: mate <file>
* **Check status**: git status
* **Commit changes**: git commit -am "Commit message"
* **Switch branches**: git checkout <branch-name>
* **Compare branches**: git diff <branch1> <branch2>
* **Visual comparison**: git difftool <branch1> <branch2>
* **Merge branches (Fast forward merge)**: git merge <branch-name>
* **Delete a branch**: git branch -d <branch-name>

**Branching without Fast forward merges**

* **Create and switch to a new branch**: git checkout -b <branch-name>
* **Check current branch**: git branch
* **Edit a file**: mate <file>
* **Check status**: git status
* **Commit changes**: git commit -am "Commit message"
* **Switch branches**: git checkout <branch-name>
* **Compare branches**: git diff <branch1> <branch2>
* **Visual comparison**: git difftool <branch1> <branch2>
* **Merge branches(Merge with no fast-forward to preserve the branch history)**: git merge --no-ff <branch-name>
* **Delete a branch**: git branch -d <branch-name>

**Automatic Merge**

**What is an Automatic Merge?**

An automatic merge in Git occurs when Git can combine changes from different branches without any conflicts. This means that Git can automatically integrate the changes from one branch into another without requiring any manual intervention.

**How Automatic Merges Work**

When you merge one branch into another, Git tries to combine the changes from both branches. If the changes do not overlap or conflict, Git can perform an automatic merge. Here’s how it works:

1. **Common Ancestor**: Git identifies the common ancestor of the two branches. This is the commit from which both branches diverged.
2. **Three-Way Merge**: Git performs a three-way merge using the common ancestor, the current branch (HEAD), and the branch being merged. It compares the changes made in both branches since the common ancestor.
3. **No Conflicts**: If there are no conflicting changes (i.e., changes made in different parts of the files or different files altogether), Git can automatically merge the branches.

**Merge Conflict**

**What is a Merge Conflict?**

A merge conflict occurs when Git is unable to automatically resolve differences between two branches. This usually happens when changes are made to the same lines of a file in both branches. When this happens, Git requires manual intervention to resolve the conflicts.

* **Create and switch to a new branch**: git checkout -b <branch-name>
* **Check current branch**: git branch
* **Edit a file**: mate <file>
* **Commit changes**: git commit -am "Commit message"
* **Switch branches**: git checkout <branch-name>
* **Merge branches**: git merge <branch-name>
* **Check status**: git status
* **Resolve conflicts manually**: Edit the file to resolve conflicts
* **Mark conflicts as resolved**: git add <file>
* **Commit the merge**: git commit -m "Resolved merge conflicts"
* **Delete a branch**: git branch -d <branch-name>

**Rebase (See from chatgpt and after that see working written below)**

**What is Git Rebase?**

Git rebase is a powerful command that allows you to integrate changes from one branch into another. Unlike merging, which creates a new commit to combine the histories of two branches, rebasing moves or “replays” your commits on top of another branch. This can result in a cleaner, linear project history.

**Why Use Rebase?**

* **Cleaner History**: Rebasing creates a straight, linear sequence of commits, making the project history easier to read and understand.
* **Avoiding Merge Commits**: Rebasing avoids the creation of merge commits, which can clutter the project history.
* **Updating Feature Branches**: Rebasing is useful for updating feature branches with the latest changes from the main branch without creating a merge commit.

**How Rebase Works**

When you rebase a branch, Git takes the commits from your current branch and applies them on top of another branch. Here’s a step-by-step explanation:

1. **Identify the Base Commit**: Git identifies the common ancestor commit between the current branch and the target branch.
2. **Apply Commits**: Git applies each commit from the current branch onto the target branch, one by one.
3. **Resolve Conflicts**: If there are conflicts, Git pauses the rebase process and allows you to resolve them

**Abort a rebase, rebase conflict and resolution, Pull with Rebase (GitHub)**

* **Aborting a Rebase:** When you start a rebase and encounter issues or decide you don’t want to continue, you can abort the rebase process. This will return your branch to the state it was in before the rebase started.

**How to Abort a Rebase**

1. **Start a Rebase**: Begin a rebase process:
   * git rebase master
2. **Abort the Rebase**: If you encounter issues or want to stop the rebase, use:
   * git rebase --abort
   * This command will stop the rebase and reset your branch to its original state before the rebase began.

**Rebase Conflict and Resolution:** During a rebase, conflicts can occur if changes in your branch and the branch you are rebasing onto affect the same lines of code. You need to resolve these conflicts manually.

**Handling Rebase Conflicts**

1. **Start a Rebase**:Begin a rebase process:
   * git rebase master
2. **Encounter a Conflict**: If there is a conflict, Git will pause the rebase and show a message like:
   * CONFLICT (content): Merge conflict in file.txt
3. **Resolve the Conflict**: Open the conflicted file in your text editor. You will see conflict markers like:
   * <<<<<<< HEAD
   * Content from master branch
   * =======
   * Content from your branch
   * >>>>>>> your-branch
   * Edit the file to resolve the conflict by choosing which changes to keep or combining them. Remove the conflict markers.
4. **Mark the Conflict as Resolved**: Stage the resolved file:
   * git add file.txt
5. **Continue the Rebase**: Continue the rebase process:
   * git rebase –continue
6. **Repeat if Necessary**: If there are more conflicts, repeat the process until the rebase is complete.

**Pull with Rebase (GitHub):** Pulling with rebase is a way to update your local branch with changes from a remote branch (like origin/master) without creating a merge commit. This keeps your commit history linear and clean.

**How to Pull with Rebase**

1. **Fetch the Latest Changes**: Fetch the latest changes from the remote repository:
   * git fetch origin
2. **Pull with Rebase**: Rebase your current branch onto the fetched changes:
   * git pull --rebase origin main
   * This command fetches the latest changes from origin/master and rebases your current branch on top of those changes.
3. **Resolve Conflicts if Any**: If there are conflicts during the rebase, resolve them as described in the rebase conflict section.
4. **Continue the Rebase**: After resolving conflicts, continue the rebase:

git rebase --continue

**Stashing**

**What is Git Stash?**

Git stash is a command that allows you to temporarily save changes in your working directory that you are not ready to commit. This is useful when you need to switch branches or work on something else without committing incomplete work.

**Why Use Git Stash?**

* **Switching Branches**: You might need to switch branches to work on something else, but you don’t want to commit your current changes.
* **Temporary Save**: You want to save your work temporarily without committing it to the repository.
* **Clean Working Directory**: You need a clean working directory to pull changes or perform other operations.

**How Git Stash Works**

When you stash changes, Git saves the current state of your working directory and staging area. You can then apply these stashed changes later.

**Basic Git Stash Commands**

**1. Stashing Changes**

1. **Stash Your Changes**: Use the git stash command to save your changes:

**git stash**

* + This command saves your changes and reverts your working directory to the last commit.

1. **Stash with a Message**: You can add a message to your stash to remember what it contains: **git stash save "Work in progress on feature X"**

**2. Listing Stashes**

1. **List All Stashes**: Use the git stash list command to see all stashes:
   * **git stash list**
   * Output:
   * stash@{0}: WIP on master: 1234567 Add new feature
   * stash@{1}: WIP on master: 89abcdef Fix bug

**3. Applying Stashes**

1. **Apply the Latest Stash**: Use the git stash apply command to apply the latest stash:
   * git stash apply
2. **Apply a Specific Stash**: Use the git stash apply command with the stash reference to apply a specific stash:
   * git stash apply stash@{1}

**4. Dropping Stashes**

1. **Drop the Latest Stash**: Use the git stash drop command to remove the latest stash:
   * git stash drop
2. **Drop a Specific Stash**: Use the git stash drop command with the stash reference to remove a specific stash:
   * git stash drop stash@{1}

**5. Popping Stashes**

1. **Pop the Latest Stash**: Use the git stash pop command to apply and remove the latest stash:
   * git stash pop
2. **Pop a Specific Stash**: Use the git stash pop command with the stash reference to apply and remove a specific stash:

* git stash pop stash@{1}

**Stashing Untracked Files and Using Pop, Managing Multiple Stashes, Stashing into a Branch**

* **Stashing Untracked Files:** By default, git stash only stashes tracked files. However, you can also stash untracked files (new files that haven’t been added to the repository) using the -u option.

**How to Stash Untracked Files**

1. **Stash Tracked and Untracked Files**:
   * Use the -u option to include untracked files:

**git stash -u**

* + This command stashes both tracked and untracked files.
* **Managing Multiple Stashes:** When you have multiple stashes, you can manage them using git stash list, git stash apply, git stash drop, and git stash pop.

**How to Manage Multiple Stashes**

1. **List All Stashes**: git stash list
   * Output:

stash@{0}: WIP on master: 1234567 Add new feature

stash@{1}: WIP on master: 89abcdef Fix bug

1. **Apply a Specific Stash**: Apply a specific stash without removing it:

git stash apply stash@{1}

1. **Drop a Specific Stash**: Remove a specific stash:
   * git stash drop stash@{1}
2. **Pop a Specific Stash**: Apply and remove a specific stash:

git stash pop stash@{1}

* **Stashing into a Branch:** Sometimes, you might want to save your stashed changes into a new branch. This is useful if you decide that the stashed changes should be developed further in a separate branch.
* **How to Stash into a Branch**

**Create a New Branch from Stash**: Create a new branch and apply the stash:

git stash branch <branch-name>

**Tagging (Asked to Shukla in interview and must need to say about versions that we can tag versions)**

**What is Tagging in Git?**

Tagging in Git is a way to mark specific points in your repository’s history as important. Typically, tags are used to mark release points (e.g., v1.0, v2.0) or other significant milestones. Tags are like branches, but they do not change. Once you create a tag, it points to a specific commit and remains fixed.

**Types of Tags**

There are two main types of tags in Git:

* 1. **Lightweight Tags:** Lightweight tags are simple references to a specific commit. They are like a branch that doesn’t change and do not store any additional metadata.

**Creating a Lightweight Tag:** Use the git tag command followed by the tag name: git tag v1.0

**Verify the Tag**: git tag

* + Output: v1.0

1. **Annotated Tags:** Annotated tags store additional metadata such as the tagger’s name, email, date, and a message. They are stored as full objects in the Git database.

**Create an Annotated Tag**: Use the git tag -a command followed by the tag name and a message: git tag -a v1.0 -m "First stable release"

**Verify the Tag**: git tag

Output: v1.0

**View Tag Details**: Use the git show command to see details of the annotated tag: git show v1.0. Output:

tag v1.0

Tagger: Your Name <your.email@example.com>

Date: Mon Oct 16 10:00:00 2023 +0000

First stable release

commit abc1234

Author: Your Name <your.email@example.com>

Date: Mon Oct 16 10:00:00 2023 +0000

Initial commit

1. **Comparing Tags:** You can compare the differences between two tags to see what changes were made between them.

**Compare Two Tags**: Use the git diff command followed by the two tag names: git diff v1.0 v2.0

This command shows the differences between the commits tagged as v1.0 and v2.0.

1. **Tagging a Specific Commit:** You can tag a specific commit by providing the commit hash when creating the tag.
2. **Tag a Specific Commit**: Use the git tag command followed by the tag name and the commit hash: git tag v1.0 abc1234
3. **Verify the Tag**: git tag

Output: v1.0

1. **Updating Tags:** If you need to update a tag, you must delete the old tag and create a new one.

**Delete the Old Tag**: Use the git tag -d command followed by the tag name: git tag -d v1.0

**Create a New Tag**: Create a new tag with the same name: git tag -a v1.0 -m "Updated release"

**Push the Updated Tag**: Push the updated tag to the remote repository: git push origin --tags

1. **Using Tags with GitHub:** Tags are useful for marking releases and important points in your project’s history. You can push tags to GitHub and manage them through the GitHub interface.
2. **Push Tags to GitHub**: Push a single tag to GitHub: git push origin v1.0
   * Push all tags to GitHub: git push --tags
3. **View Tags on GitHub**: Go to your repository on GitHub and navigate to the “Tags” section to see all the tags.
4. **Delete a Remote Tag**: Delete a tag from the remote repository: git push --delete origin v1.0