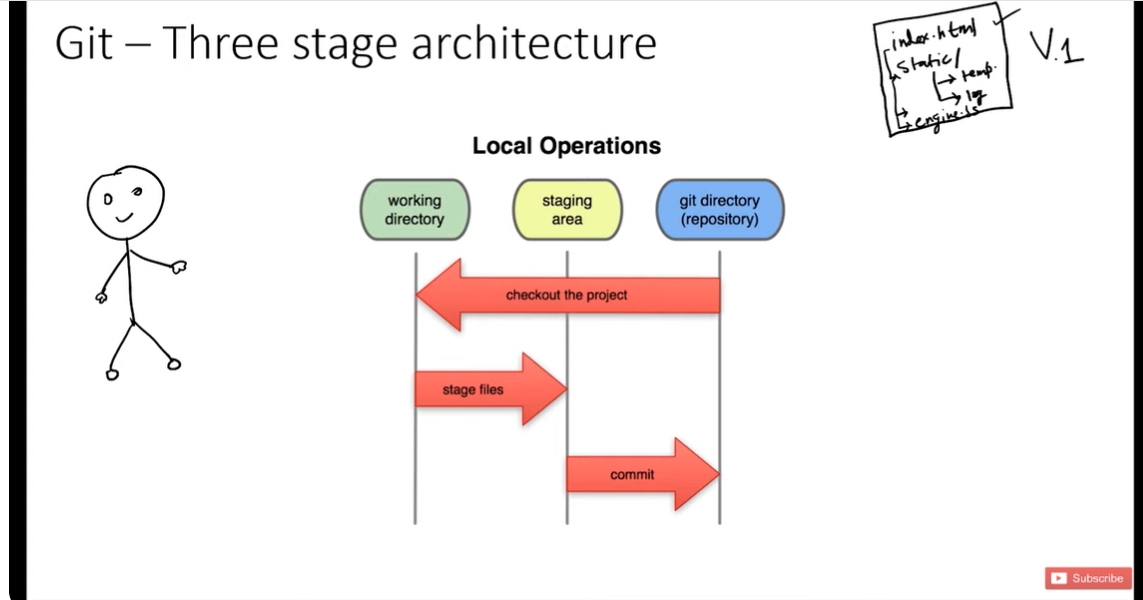
Git is a Distributed version control system that lets developers track source code changes during software development. Git Bash is an application for Microsoft Windows, allowing developers to use Git in a command-line interface.

Gitbash , a terminal more like linux used for Git command line experience. Bash is an acronym for Bourne Again Shell. Where Bash means limit/boundary

Gitignore >> Files or directories listed in gitignore will not be tracked and ignored every time and will not show up on git terminal in untracked section.

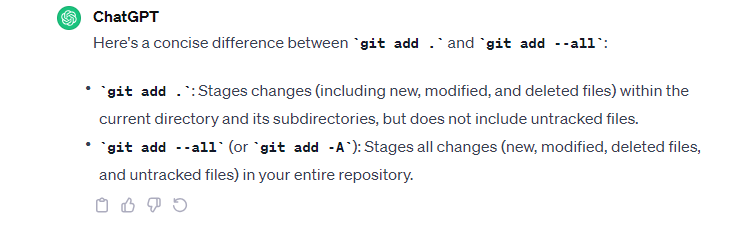
**Git three Stage Architecture:**

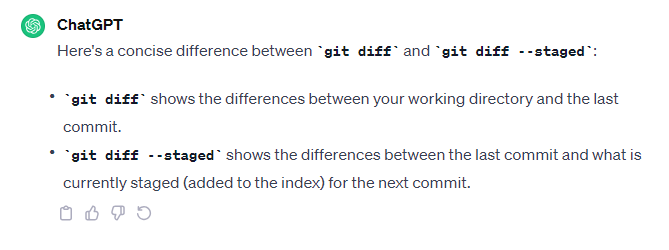


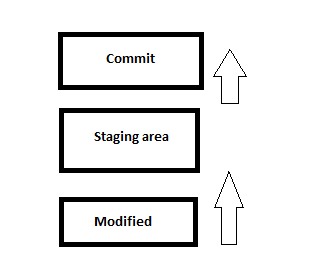
**File Status Lifecycle**



**Git add . vs git add - -a**







|  |
| --- |
| **Commands:**   1. git config –-list //shows git configuration settings 2. git config --global user.name "Atisam" 3. git config --global user.email "atisam82@gmail.com" 4. git status // show current status: modified files/staged/tracked/untracked 5. git add - - a // stage all files for commit in repository 6. git add “filename” //stage only file.txt for commit not all files. E.g. git add “file sn.txt” 7. git commit -m “first commit” //snapshot taken in local folder/repository 8. git log //check commits history of **current branch only** //switch to see others’ history 9. git diff //compare working directory/modified and staging area (ready for commit) 10. git diff --staged //compare last commit vs current staging area 11. git rm file1.txt //remove file.txt and also stage it as deleted, if I manually delete file, then I have to git add – - a for staging 12. git mv file1.txt file1\_rename.txt //rename file and stage it automatically 13. git restore --staged file3.txt //unstage file with this command 14. git restore file3.txt //undo changes in file 15. git checkout -- file3.txt //undo changes in file 16. git checkout –f // undo changes of all files 17. git fetch origin // just tells changes in remote repository like commits. After this use git status 18. git merge origin/main //same like above command but little more detailed view. 19. git pull origin main // add all latest changes from remote address directly   // after this use “git status”, now local repo is up-to-date   1. git config --global alias.s "status" //set alias(short) command now “git s” is same as “git status” set more alias what u want   **git config –global alias.a “add --a” // git a = git add –a**  **git config –global alias.c “commit -m” // git c “msg”=git commit -m “msg”**   1. git branch -d new //delete branch if merged 2. git branch -D new //delete branch if not merged (forceful) 3. git push -d origin new //delete remote branch new 4. git branch -v // shows all branches with last commits made 5. git branch –merged // **list all local branches that have been fully merged into the current branch** 6. git branch --no-merged //list all local branched that haven’t been fully merged   //you need to not to be in same branch which are u deleting |

|  |
| --- |
| **remote Commands (for working on starter):**   1. When “git push -u origin main”, git will ask for GitHub login automatically |

|  |
| --- |
| **remote Commands (for working on existing remote repository specifically for multiple collaborators):**   1. git remote add origin ”<https://github.com/atisamhaq123/git.git>” //add remote origin 2. git remote -v // view origin remote address 3. git branch -M main // rename the current branch to "main">> before pushing code to remote repo having this branch name // “main” it's necessary to rename your local computer branch as main to streamline workflow 4. git fetch origin // view what’s new in remote repository by fetching changes. **Just view** 5. git merge origin/main // **ignore** same like above command but more detailed view. **Just view** 6. git pull origin main // make your local repo up-to-date by **adding** all new changes from remote   // use “git status” and your repo will be up-to-date   1. **Do your changes (then> git status >>git add --a >>git commit -m “commit” >>git status >>** 2. git push -u origin main //push to main branch of remote repository and   **//branch is parallel space for development in repository useful for bug fixing, conflict resolution and test/review**  **//before switching branch, make sure files you have commit and branch is up-to-date**   1. git checkout -b develop //create develop branch and switch to develop 2. git checkout develop // switch to develop just 3. git branch // view all branches 4. git merge “develop” // merge develop branch into main branch but make sure current branch is main by switching: git checkout main and then “git merge develop” : now changes of **develop branch will merge into main** and VS code will allow to accept incoming or new changes HENCE, your choice can easily committed     **Summary:** |

|  |
| --- |
| **Pro Commands:**   1. git commit -a -m “direct commit” //skip staging area and direct commit but untracked files which are not in staging area will not be committed 2. git rm --cached file3.txt //file stays in working directory but git doesn’t track it further. For git, this file is deleted 3. rm -rf .git //now it’s not git repository anymore 4. git log -p //shows what added or removed in commits. Use arrow up/down key to view in detail 5. git log -p -2 //shows what added or removed in last 2 commits. Use arrow up/down key to view in detail 6. git log –-stat //shows changes in commit in short form/summary 7. git log --pretty=oneline //shows changes in commits in **1 Line** 8. git log --pretty=short //see changes in short 9. git log --pretty=full //see changes in full 10. git log --since=100.days //see changes of last 100 days 11. git log --since=10.weeks //see changes of last 10 weeks 12. git log --since=3.months //see changes of last 3 months 13. git log --since=2.years //see changes of last 2 years 14. git log --pretty=”ah -- an” //see changes of last 100 days 15. git log --pretty=format"%h -- %an" //format commits with tag – author name 16. git log --pretty=format"%h -- %an" //format commits with tag – author email 17. git reset --soft HEAD~1 // undo the last commit but keep the changes in your staging area 18. git reset --hard HEAD~1 // undo the last commit and completely discard the changes made in that commit   **//use git log to check commit hash**   1. git reset --hard commit\_hash //e.g., git reset --hard e86c6defe46b57ee89cec9bd226ba5f91a0be4ea undo all changes after that commit 2. git reset --hard commit\_hash //e.g., git reset --soft e86c6defe46b57ee89cec9bd226ba5f91a0be4ea undo all changes after that commit but nothing deleted in fact keep your changes from commits after it as staged changes |

|  |  |
| --- | --- |
| **CLI access Two ways:**   1. Temporary: Access GitHub through browser session token, first time you use gitbash using “git remote” 2. Another way to use SSH(Secure Shell Protocol) by permanently add SSH key in Github settings  |  | | --- | | **Commands:**   * ssh-keygen -t ed25519 -C [atisam82@gmail.com](mailto:atisam82@gmail.com) //generate fingerprint * eval "$(ssh-agent -s)" // view agent PID * ssh-add ~/.ssh/id\_ed25519 // added ssh key in windows * cat ~/.ssh/id\_ed25519.pub // view key now   **//paste this key in GitHub SSH settings**   * ssh -T [git@github.com](mailto:git@github.com) // verify and Test authentication   **//Now u can use git pull and push request through git cli i.e., gitbash** | |

**Random:**

1. git add - -a for staging >> next git commit -m “commit”
2. before staging u can check what is modified using git diff

|  |
| --- |
| 1. >> touch .gitignore 2. >> add directory name which u want to ignore like >>>>>> log/ 3. >> add extension of files which u want to ignore >>>>> \*.log 4. >> Now these files or directories will not be tracked and ignored every time and will not show up on git terminal in untracked section |

1. pwd: command shows present working directory
2. ls: command list the files and directories in the current directory
3. <<< >>> called conflict resolution markers, comes when merge two branches

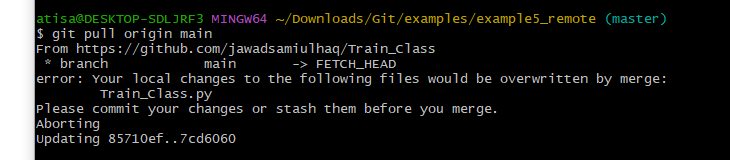
**Scenario:**

**How will u make sure u are committing files u changed?**

First make sure u are modifying files that u r supposed to do. In staging area: your files must be there using command “git add filename.txt”. If you accidently modified wrong file, don’t worry u can undo changes or u can make sure it’s not in staging area. For commit and later for push, only files present in staging area are important. So, keep looking what files are staged. Another thing is you can add unnecessary files in .gitignore to left it untracked

**Always push local changes(if modified) before pulling new changes otherwise conflict**

* First, we modified, then we staged files for commits and before pushing to remote, we pulled latest changes, this will abort and It is conflict. To solve it, first push your local changes and then pulling new ones from remote address.



**Even if files are permanently deleted SHIFT+D, git commit can revert that if they were available in commit history**

**Take it easy?**

Unless u don’t change a file, it’s not eligible to go in staging area and later for commit and then finally for push. So, if u pulled a repository your job is to change design. Your fellow is working on backend files, make sure u don’t change that backend files and modify only design files. After improving design, u can stage specific modified files or all modified files using “git add filename.txt” or “git add --a” respectively.

Modified files will be staged for commit now and then push

**Branching and Merging Scenario**

1. git checkout -b new //creates a branch and switch to that branch
2. Do some changes and commit in new
3. Next checkout to main branch: **git checkout -b main**
4. Finally, **git merge new //** merge new branch to main

|  |
| --- |
| **What is Conflict:**  A conflict occurs when Git is unable to automatically merge changes from one branch into another because there are conflicting changes in the same part of the same file. This typically happens when:   1. In one branch, a line is modified (changed) in a file. 2. In another branch, the same line in the same file is also modified, but the changes are different.   In this situation, Git doesn't know which change to apply, and it marks the file as having a conflict. It's then up to the developer to manually resolve the conflict by choosing which version of the change to keep or by making further modifications to reconcile the differences.  **What’s NOT Conflict:**  On the other hand, if one branch adds lines or remove lines to a file and another branch modifies different lines in the same file, Git can automatically merge those changes without conflict because they don't overlap. Git can simply combine the additions and modifications from both branches. |

**>> Considering only Modifications Case now (like same line modifies) <<**

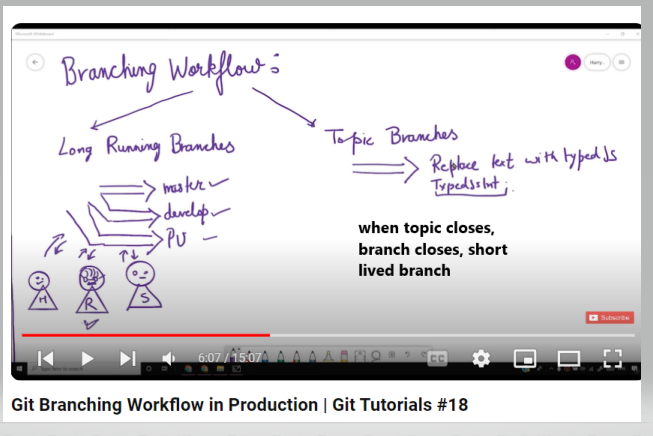
**Scenario1(NO CONFLICT):**

|  |
| --- |
| 1. You commit changes in the main branch. 2. You create a new branch and make additional commits. 3. You switch back to the main branch. 4. You merge the new branch into the main branch.   In this scenario, there **are no conflicting** changes between the main branch and the new branch, the merge process will be smooth, and there won't be any conflicts. Git will automatically merge the changes from the new branch into the main branch.  **Description:**   * C0, C1, C2 commits in main branch: * Next, we switch to new branch and do C4, C5, C7 commits * Next, we switch back to main branch * Next, we merge and here is a no conflict 😊 |

**Scenario2(Conflict):**

|  |
| --- |
| 1. You commit changes in the main branch. 2. You create a new branch and make additional commits. 3. You switch back to the main branch. 4. You make more commits in the main branch. 5. You attempt to merge the new branch into the main branch.   In this scenario, if there are **conflicting** changes between the main branch and the new branch  **solution: VS code allows you to accept which changes u want, current or incoming 😊**  **Example:**    **Description:**   * C0, C1, C2 commits in main branch: * Next, we switch to new branch and do C4, C5, C7 * Next, we switch back to main branch and **commit C6** * Next, we merge and here is a conflict 😊   **Output:**    **Solution**: Use VS code to allow what changes u want |
| After accepting incoming changes, run gitbash, type “git status”  now see, no conflicts no but just unmerged paths    Run command “**git add --a**”, now unmerged paths and every other change is staged.    Run command **git commit -m “done**”    See changes from new branch are successfully merged |

**Branching Work-Flow**



Live scenario

|  |
| --- |
| Here's a sequence of Git commands along with one-line descriptions for each command in the provided flow:   1. **git init**: Initializes an empty Git repository in the current directory. 2. **git remote add origin https://github.com/atisamhaq123/git.git**: Adds a remote named "origin" with a URL to a GitHub repository. 3. **git remote -v**: Lists the configured remote repositories and their URLs. 4. **git branch -M main**: Renames the current branch from "master" to "main." 5. **git fetch origin main**: Fetches changes from the "main" branch of the remote repository named "origin." 6. **git fetch origin**: Fetches changes from all branches in the remote repository. 7. **git pull origin main**: Pulls changes from the "main" branch of the remote repository "origin" into the current branch. 8. **git commit -a -m "msgs"**: Commits changes with a commit message. 9. **git add --a**: Stages all changes (including untracked files) for the next commit. 10. **git checkout -b new**: Creates a new branch named "new" and switches to it. 11. **git commit -a -m "msgs"**: Commits changes in the "new" branch with a commit message. 12. **git checkout main**: Switches back to the "main" branch. 13. **git push -u origin new**: Pushes the "new" branch to the remote repository and sets up tracking. 14. **git merge new**: Merges changes from the "new" branch into the "main" branch. 15. **git status**: Displays the current branch's status, including any pending commits. 16. **git push -u origin main**: Pushes changes from the "main" branch to the remote repository and sets up tracking. 17. **git branch**: Lists all branches in the repository. 18. **git push -d origin new**: Deletes the "new" branch from the remote repository. 19. **git branch -D new**: Deletes the "new" branch locally |

**Note:**

**Below commands just tell changes, doesn’t mean they actually bring changes in local repository. For that we need to pull like: git pull origin main**

1. **git fetch origin // tells changes in overall remote repository**
2. **git fetch origin main //tells changes in main branch**

**Work Specific:**

|  |
| --- |
| 1. git fetch origin branchname //see changes 2. git merge FETCH\_HEAD //merge   **//FOLLOW here, OTHER STEPS: git status, git add. git commit.**   1. git push origin your\_branch\_name //push new changes |

**For conflict Scenario, if merge doesn’t take place**

|  |
| --- |
| 1. **Git Status:** After running the **git merge FETCH\_HEAD** command and encountering conflicts, you'll see a message indicating that there are conflicts. You can also use **git status** to see which files have conflicts.   git status   1. **Resolve Conflicts:** Open the conflicted files in your text editor. You'll see markers in the file indicating the conflicting changes. You need to manually edit the file to resolve the conflicts. Remove the conflict markers (**<<<<<<<**, **=======**, and **>>>>>>>**) and keep the changes that you want to keep. 2. **Mark as Resolved:** After resolving the conflicts in each conflicted file, mark them as resolved using:   git add <conflicted\_file>it add <conflicted\_file>  Replace **<conflicted\_file>** with the path to the conflicted file.   1. **Continue Merge:** After resolving all conflicts and marking the conflicted files as resolved, you can continue the merge using:   git merge --continue  This command finalizes the merge process using the resolved changes.   1. **Commit the Merge:** Once you've resolved the conflicts and continued the merge, Git will prompt you to enter a commit message for the merge:   git commit commit  This command opens your default text editor for you to enter a commit message. Alternatively, you can use **git commit -m "Your merge commit message"** to commit with a message directly from the command line.   1. **Push Your Changes (if needed):** If you want to push your merged changes to the remote repository, use the **git push** command:   git push origin your\_branch\_nameorigin your\_branch\_name |