1. **Command to configure and check user name in Git.**

   Ans -> git config --global user.name <your_name> -----> for configuring

   git config user.name ----> for checking

2. **Command to cofigure and check email in Git.**

   Ans -> git config --global user.email <your_email> -----> for configuring

   git config user.email ----> for checking

3. **How to delete a git repo?**

   Ans- -> To delete a git repository, we need to delete the '.git/' folder from the repo,

   so first go inside the repo you want to delete and then run this command: rm -rf .git/

4. **How to make a repository?**

   Ans -> To create a new repository, we need to run the git init -> this command inside the directory

   which we want to make a repository.

   Note:-> *Do not run this command inside an existing repository, before running 'git init', use 'git status' to verify that you are not currently inside of a repo.*

   git status -> this command gives information on the status of a git repository and its
contents.

5. **How to add files to the staging area in git?**

   Ans -> git add -> this command is used to add specific files to the staging area. Separate file with spaces to add multiple at once. eg.- $git add file1 file2

   if you want to stage all the modified files at once, then use this command -> git add .

6. **How to commit staged changed in git?**

   Ans -> git commit -> this command will commit all staged changes. it also opens up a text editor (eg. VScode) and prompts you for a commit message, or if you want to give the message along with the command then use this syntax: git commit -m "my message".

7. **How to track or get information about all the commits that have been made so far?**

   Ans -> git log -> this command is used to track or give information about all the commits that have been made so far.

   git log --oneline -> this command is used to display a simplified and condensed version of the commit history in a Git repository. When you run this command, each commit is shown on a single line, providing a quick overview of the commit information.

8. **What does 'code .' command do?**

Ans -> code . -> this command will open your current directory in VScode.

9. **What is Atomic Commit?**

Ans-> An Atomic Commit is like making a single package of changes to your project. Instead of changing many things at once, you group related changes together. It's a way of organizing your work, so each package (or commit) has a clear purpose. This makes it easier for you and your team to understand, review, and manage the progress of your project. If something goes wrong, you can fix or undo just that package without affecting everything else. It's a way of keeping your project clean and tidy.

10. **What does 'git commit --amend' command do?**

Ans -> git commit --amend -> this command is used to modify the most recent commit in a Git repository. When you run this command, it allows you to make changes to the commit message, add more changes to the previous commit, or both. Essentially, it combines the staged changes with the changes in the previous commit and creates a new commit with an updated message.

Here's how it works:

1. First, you make additional changes to your files and stage them using git add

2. Run git commit --amend: This command will open your default text editor, displaying the commit message of the previous commit and allowing you to modify it. You can make changes to the message or leave it as is.

3. Save and close the editor: After making changes (if any) to the commit message, you save and close the text editor. Git will then create a new commit that includes the changes from the previous commit and the additional changes you staged.

11. **What is .gitignore?**

Ans -> The .gitignore file is a configuration file used in Git to specify files and directories that should be ignored by version control. When you're working on a project, there are often files or directories that you don't want Git to track. These might include temporary files, build artifacts, configuration files with sensitive information, or any other files that are generated during the development process.

The .gitignore file allows you to list such files and patterns so that Git knows to disregard them when you commit changes. This helps keep your repository clean and prevents irrelevant or sensitive files from being included in the version history.

12. What are branches in git?

Ans -> In Git, branches are divergent lines of development that allow developers to work on separate features, bug fixes, or experiments simultaneously. The main branch, often named "master" or "main," represents the stable project state. Feature branches isolate changes for specific features, while release and hotfix branches assist in preparing and fixing code for deployment. Developers switch between branches, create, and merge them to manage and organize collaborative development effectively, preventing conflicts and enabling efficient version control.

In simple words, branches are like alternative timelines of a project.

They enable us to create separate contexts where we can try new things, or even work on multiple ideas in parallel.

If we make changes at one branch, they do not impact the other branches (unless we merge the changes).

13. What is HEAD in git?

Ans -> In Git, HEAD is like a pointer that shows you where you are in your project. It points to the latest snapshot of your work, usually the most recent commit on the branch you are currently working in. It helps Git know which version of your project you are looking at and where new changes will be added.

14. How to check and create branches in git?

Ans -> The `git branch` command in Git is used to list, create, or delete branches within a Git repository. When you run `git branch` without any arguments, it shows a list of existing branches and indicates the currently active branch with an asterisk (*).

`git branch <branch_name>` -> use this command to create new branch.

`git switch <branch_name>` -> use this command to switch between branches.

`git branch -v` -> this command in Git shows a list of branches along with some additional information about each branch. Specifically, it displays the commit hash (SHA-1) and the commit message that is at the tip of each branch. The -v option stands for "verbose."

15. How to stage and commit changes at the same time in git?

Ans -> The git commit -a -m <my_message> or git commit -am <my_message> command in Git combines two actions: it stages all modified files and commits them with a specified message.

This command is useful when we want to quickly commit all the changes that we've made to tracked files in our working directory without staging each file individually. However, it's important to note that *this command does not include untracked files*; we still need to use git add for new files.

16. What are the different ways to create and switch branches at the same time?

Ans -> git switch -c <branch_name> -> use this command to create and switch to a new branch at the same time.

we can also use git checkout -b <branch_name> -> this as an alternative, it will do the same as previous one.

17. How to rename a branch?

Ans -> To rename a Git branch, first, ensure you are on a different branch than the one you want to rename. Then, use the command git branch -m <old_branch_name> <new_branch_name> to rename the branch locally.

And if you want to rename a branch which you are currently on, then use:
git branch -m <new_branch_name>

18. How to delete a branch?

Ans -> To delete a branch in Git, use git branch -d <branch_name> or to forcefully delete a branch use git branch -D <branch_name> command.

Note1: *You cannot delete a branch with git branch -d <branch_name> this command, if it contains changes that haven't been merged into the branch you are currently on. This is to avoid unintentional data loss.*

Note2: *You cannot delete the branch you are currently on. If you are on the branch you want to delete, switch to another branch before attempting the deletion.*

19. What is merge in git?

Ans -> In Git, merging is like combining different sets of changes made by different people into one shared version of the code. Imagine each person works on their own version (branch) of the project. When they finish something, they merge it back into the main project. Git helps put everything together smoothly, but sometimes people might change the same thing, causing a conflict that needs manual fixing. Once everything fits well, the changes become part of the main project that everyone can use.

To merge follow these basic steps:

1. Switch to or checkout the branch you want to merge the changes into (the receiving branch)

Example: `git switch master`

2. Use the git merge command to merge changes from a specific branch into the current branch.

Example: `git merge <branch_name>`

20. What is Fast-Forward merge in git?

Ans -> A fast-forward merge in Git is a merging strategy where the branch being merged has all the new commits in its history that the branch it is merging into has. This type of merge is called "fast-forward" because Git can simply move the pointer of the branch you are merging into to the latest commit of the branch you are merging, without the need for a new merge commit.

You have a branch, let's call it "feature-branch," where you made some changes.

The branch you want to merge into, typically the main branch (e.g., "master"), has not diverged since you created "feature-branch."

To perform a fast-forward merge:

```
# Make sure you are on the branch you want to merge into (e.g., master)
git switch master

# Merge the feature branch (assuming it's named "feature-branch")
merge feature-branch
```

21. What is merge conflicts and how to resolve them in git?

Ans -> In Git, a merge conflict occurs when there are conflicting changes in different branches that Git is trying to merge. This typically happens when two branches have modifications to the same part of a file, and Git cannot automatically determine which changes to incorporate. Resolving merge conflicts involves manually selecting and combining the conflicting changes before completing the merge.

<u>Wherever you encounter merge conflicts, follow these steps to resolve them:</u>

1. Open the file(s) with merge conflicts.

2. Edit the file(s) to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.

3. Remove the conflict "markers" in the document.

4. Add your changes and them make a commit.

22. What are the conflict markers in a file?

Ans -> The conflict markers in a file are displayed as below:

```
<<<<<<<<<<< HEAD

file contents

==========

file contents to merge

>>>>>>>>>>> bug-fix
```

The content from your current HEAD (the branch you are trying to merge content into) is displayed between the <<<<<<<< HEAD and =========

The content from the branch you are trying to merge from is displayed between the ========= and >>>>>>>>> symbol

23. What does git diff command do? (Working directory VS Staging area)

Ans -> The git diff command in Git is used to display the differences between various states of a Git repository. It can compare changes in the working directory with the staging area, the staging area with the last commit, or the working directory with the last commit. Additionally, it can compare between two specific commits. The command outputs the lines that have been added, modified, or deleted, with plus and minus signs indicating additions and deletions, respectively. It's a versatile tool for reviewing and understanding changes in the codebase.

git diff -> this command without any additional options, lists all the changes in out working directory that are not staged for the next commit.

git diff HEAD -> Git checks what's different between:

The files in your current project as you've changed them (but haven't committed yet), and The last saved version of those files (the last commit you made).

So, git diff HEAD tells you what you've changed since your last commit. It's like asking Git to highlight the alterations you've made to your project since the last time you saved your work.

git diff --staged or **git diff --cached** -> When you run git diff --staged or git diff --cached, Git shows you the changes you've already added to the staging area (using git add) but haven't yet committed.

It's like asking Git to compare what's ready to be committed with what's already been saved in the last commit. This helps you review your changes before making them permanent with a commit.

git diff branch1 branch2 or **git diff branch1..branch2** -> this command is used to differentiate between 2 branches, that is branch1 and branch2.

It's like comparing two versions of your project to see what's changed between them. This helps you understand the differences between the two branches before merging them or making other decisions based on those changes.

git diff commit1 commit2 --> this command is used to differentiate between 2 different commits.

It's like comparing two pictures of your project at different points in time to see exactly what changed between them. This helps you understand the specific alterations made between those two moments in your project's history.

24. What git stash?

Ans -> **git stash** -> this is a very useful command that helps you to temporarily store changes that are not ready to be committed yet. It allows you to save your modifications, revert your working directory to a clean state, and then reapply the changes later when needed. This can be useful when you need to switch branches or pull in change from another branch without committing your current changes.

Note: *Running git stash command will take all uncommitted changes (staged and unstaged) and stash them.*

git stash pop -> This command is used to retrieve the most recent change you stashed away using git stash command. It not only brings back those changes but also removes them from the stash. It's like taking your stashed changes out of the drawer and putting them back into your working directory.

git stash apply -> This command is similar to git stash pop, but it doesn't remove the changes from the stash after applying them. It simply applies the most recent stash onto your working directory, allowing you to keep the changes in the stash for future use. It's like a copy of your stashed changed and applying them to your current work without removing them from the stash drawer. This can be useful if you want to apply stashed changes to multiple branches.

git stash list -> This command is used to display a list of stashed changes in your repository. It shows you the stash entries along with a unique identifier (often a hash) for each stash, making it easier to reference or manage multiple stashes. It's like peeking into the stash drawer to see what you've stored away.

Examples:

```
$> git stash list

stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
```

- To apply or use a specific stash, use the command **git stash apply <stash-id>**
- To delete a particular stash, you can use **git stash drop <stash-id>**
- To clear out all stashes, run **git stash clear**

Note: *By default, if you run git stash apply or git stash pop, git assumes you want to apply or pop the most recent stash.*

Examples:

To apply -> git stash apply stash@{2} or git stash pop stash@{No.}

To delete -> git stash drop stash@{2} or git stash drop stash@{No.}

25. How to go back to a previous commit?

Ans -> The git checkout <commit-hash> or git checkout HEAD~<number> command is used in Git to switch the current branch to a specific commit identified by its hash. It's like moving to a specific moment in the history of your project and seeing how your project looked at that point of time.

When you use the above command, Git will:

- Move the HEAD pointer to point to the specified commit.

  Note: HEAD do no point to any particular commit, HEAD points to a particular branch and branch points to a commit.

- Update the working directory to reflect the state of the repository at that commit.

- Put you in a "detached HEAD" state, meaning you are no longer on a branch but directly on a specific commit.

You can inspect, test, or work on that version, but remember, *you're not on any particular branch; it's like being in a standalone mode*. If you want to make changes, it's safer to create a new branch from that commit using git checkout -b <new-branch-name> or git switch –c <new-branch-name>

To exit from “detached HEAD” state, just switch back to the previous/any branch or if you want to make any changes from detached HEAD state then just create and switch to a new branch from that point and you're good to go.

26. What is the shortcut to switching back to the previously checked out branch?

Ans-> git switch - -> This command is used to switch back to the previously checked out branch in Git. It's shorthand for switching to the branch you were on before switching to the current branch. This can be convenient if you need to quickly switch between branches without specifying the branch name each time.

27. How to discard changes in git?

Ans ->

git checkout HEAD <file_name> and git checkout -- <file_name> -> both these commands are used to discard changes made to a specific file in your working directory and replace it with the version of the file from the last commit. However, they have slightly different nuances:

git checkout HEAD <file_name>: This command retrieves the version of <file_name> from the last commit (HEAD) and overwrites the changes in your working directory with that version.

git checkout -- <file_name>: This command also reverts <file_name> to its state in the last commit, but it's a more generic form of the checkout command. The -- serves to separate the <file_name> from any branch or commit references that might be mistaken for file names. This one is more concise and widely used.

git restore <file_name> -> This command is used to discard changes made to a specific file in your working directory and restore it to the state it was in at the last commit. It's a convenient way to undo modifications you've made to a file that you haven't staged for commit yet. Essentially, it's like saying, "Hey Git, I messed up this file, can you make it look like it did before?"

NOTE: The above command is not "undoable", if you have uncommitted changes in the file, they will be lost!

git restore --source HEAD~<number> <file_name> -> This command is a bit more advanced. It allows you to restore a file to a specific state from a previous commit in the repository history. Here, HEAD~<number> (here, you can use commit hash as well) refers to a commit relative to the current HEAD position, and <file_name> is the name of the file you want to restore.

This command is useful when you want to revert a file to a state that's not necessarily the last commit but a few commits back. It's like saying, "Hey Git, can you make this file look like it did X commits ago?"

28. How to unstage files from the staging area in git?

Ans -> The git restore --staged <file_name> command in Git is used to unstage changes for a specific file, removing them from the staging area while leaving the modifications intact in the working directory. This allows users to selectively exclude changes from the next commit without discarding them entirely.

29. How to reset a branch back to a particular commit in git?

Ans -> The git reset <commit-hash> or git reset HEAD~<number> command reset a branch back to a particular commit in history, basically it moves the current branch pointer to a specific commit, typically denoted by its unique hash.

When you use this command, Git adjusts the HEAD to point to the specified commit, effectively resetting the current branch to that commit. However, the changes between the previous state and the specified commit are preserved in the working directory, allowing users to recommit, modify, or discard them as needed. This command is often used to undo commits or to rework the commit history in a repository.

On the other hand, if you want to reset the branch back to a particular commit in history as well as want to get rid of all the changes between the previous state and the specified commit then use the command : git reset --hard <commit-hash> or git reset --hard HEAD~<number> .

30. What does git revert <commit-hash> command do?

Ans -> The git revert <commit-hash> command in Git is used to create a new commit that undoes the changes introduced by a specific commit identified by its hash. When you execute this command, Git will:

- Automatically create a new commit that reverses the changes introduced by the specified commit.
- Add this new commit to the current branch, effectively undoing the changes introduced by the original commit.
- Keep the commit history intact by adding a new commit that acts as an "undo" for the changes made in the specified commit.

- This is a safer way to undo changes compared to commands like git reset, as it preserves the commit history and avoids rewriting the repository's history. It's particularly useful when you need to revert to a specific commit while keeping a record of the reversion in the commit history.

31. How to clone remote repositories in git?

Ans -> The git clone <url> command in Git is used to create a copy of an existing Git repository from a remote location, such as a repository hosted on a server or a service like GitHub, GitLab, or Bitbucket. When you execute this command, Git:

- Downloads the entire repository, including all branches, commits, and files, from the specified URL.

- Sets up a local copy of the repository on your machine, including all the necessary Git metadata.

- Sets the remote URL of the repository to the URL from which it was cloned, allowing you to fetch updates from and push changes to the original repository.

- In simpler terms, git clone <url> creates a local copy of a Git repository on your computer, making it easy to collaborate on or work with the codebase.

32. How to manage remote repositories?

Ans -> The git remote command in Git is used to manage remote repositories. When you run git remote, it typically shows a list of remote repositories that your local repository is aware of. These remote repositories are typically URLs where your local repository can send changes (push) or retrieve changes (fetch/pull).

Common subcommands of git remote include:

git remote add <name> <url>: This command is used to add a new remote repository to your local Git configuration. <name> is a label you assign to the remote repository, and <url> is the URL of the remote repository.

git remote remove <name>: This command removes a remote repository from your local Git configuration.

git remote -v: This command lists the remote repositories along with their corresponding URLs, which is particularly helpful for seeing the details of each remote.

33. How to push work to GitHub?

Ans -> The git push <remote> <branch> command in Git is used to push the changes in a specific local branch to a remote repository. Here's how it works:

<remote> specifies the name of the remote repository where you want to push your changes. This is typically the label you assigned to the remote repository when you added it using git remote add.

<branch> specifies the local branch whose changes you want to push to the remote repository. Git will push the commits made to this branch to the specified remote repository.

For example, if you want to push the commits from your local branch named "main" to a remote repository named "origin", you will use the command git push origin main.

This command is crucial for collaborating with others and keeping remote repositories up to date with your local changes.

git push <remote> <local-branch>:<remote-branch> -> This command in Git is used to push changes from a local branch to a specific branch on a remote repository (the local branch name and the remote branch name can be different).

git push -u <remote> <branch> -> This command in Git is used to push local commits to a remote repository and simultaneously set up tracking so that the local branch is linked with the remote branch.

Here the -u flag stands for "upstream." When you use it, Git sets the upstream branch for the specified local branch. This means that in future pushes or pulls, Git will automatically use the tracked remote branch without needing to specify it explicitly.

For example, if you're pushing changes from your local "main" branch to a remote repository named "origin", you would use the command git push -u origin main.

This command is particularly useful for establishing the relationship between local and remote branches, streamlining future interactions between them.

34. What does git branch -r command do?

Ans -> The git branch -r command lists all the remote branches in the Git repository. Remote branches are branches that exist on the remote repository (like GitHub, GitLab, Bitbucket) but haven't been fetched or checked out locally.

When you run git branch -r, Git will display a list of these remote branches, prefixed with the name of the remote they belong to, such as origin/branch_name where origin is the default name for the remote repository. This command is useful for seeing what branches exist on the remote repository and for tracking changes in the remote branches.

35. In a remote repository multiple branches are there, and by cloning that repo locally, I got a default local branch name 'main' or 'master', but I want to work on the other branches which are available in remote repository but not listed locally. What should I do to achieve this?

Ans -> Run git switch <remote-brach-name> to create a new local branch from the remote branch of the same name.

Example:

git switch puppies -> This command make me a local puppies branch AND sets it up to track the remote branch origin/puppies.

36. What is Git Fetching?

Ans -> Git fetching is the process of retrieving changes from a remote repository (such as GitHub, GitLab, Bitbucket) and updating the corresponding remote-tracking branches in your local repository. When you fetch, Git will bring down any new branches or updates to existing branches from the remote repository, but it won't automatically merge those changes into your current working branch.

It lets you see what others have been working on, without having to merge those changes into your local repo.

In other words, it's like checking your mailbox for new letters but not reading them immediately.

37. How to fetch new changes from remote repository to local repository?

Ans -> git fetch <remote> -> This command fetches all the new branches and updates from the remote repository into your local repository. It doesn't merge these changes into your current branch.

git fetch <remote> <branch> -> this command fetches only the specific <branch> from the remote repository into your local repository. Again, it doesn't automatically merge these changes into your current branch.

Note: *By fetching the new changes, the only branch that will reflect the new changes is the remote-tracking branch named : 'origin/<branch_name>', so if you want to see the new changes then you can detach your head and point that head to 'origin/<branch_name>' by using the command :*

git checkout origin/<branch_name>

38. What is pulling in git?

Ans -> Pulling in Git is the process of retrieving changes from a remote repository and integrating them into your local branch. When you execute git pull, Git first fetches the latest changes from the remote repository and then automatically merges them into your current local branch.

This ensures that your local branch stays updated with any modifications made by other collaborators on the remote server, facilitating seamless collaboration and synchronization in Git workflows.

It's like, ***git pull = git fetch + git merge***

39. How to pull changes from remote repo to local repo?

Ans -> To pull we specify the particular remote and branch we want to pull using the command git pull <remote> <branch>. Just like with git merge, *it matters WHERE we run this command from*.
Whatever branch we run it from is where the changes will be merged into.
git pull origin master -> would fetch the latest information from the origin's master branch and merge those changes into our current local branch.

40. What will happen if we run only just git pull command?

Ans -> If we run git pull without specifying a particular remote or branch to pull from, git assumes the following:

- remote will default to origin
- branch will default to whatever tracking connection is configured for your current branch

Note: *this behavior can be configured, and tracking connections can be changed manually.*

41. What are the difference between git fetch and git pull command?

Ans ->

git fetch -> Gets change from remote branch(es)
git pull -> Gets changes from remote branch(es)

git fetch -> Updates the remote-tracking branches with the new changes
git pull -> Updates the current branch with the new changes, merging them in

git fetch -> Does not merge change onto your current HEAD branch
git pull -> It does, but can result in merge conflicts

git fetch -> safe to do at anytime
git pull -> Not recommended if you have uncommitted changes!

42. What is Markdown in GitHub?

Ans -> Markdown in GitHub is a simple way to format text. It lets you make *text bold*, *add headers*, *create lists*, *insert links* and *images*, and more, using easy-to-remember symbols.

You can use Markdown to write README files for your projects, format comments in issues and pull requests, and create documentation in wikis. It's a quick and efficient way to make your text look nice without needing to know HTML or use complicated tools.

For example, you must visit this link -> *https://markdown-it.github.io/*

43. What are GitHub Gists?

Ans -> Gists in GitHub are like online sticky notes where you can save and share small pieces of code, text, or markdown. You can make them public or private, share them with others, and even embed them in websites. They're handy for sharing quick code snippets or notes without needing to create a full-blown repository.

44. What are GitHub Pages?

Ans -> GitHub Pages is a feature of GitHub that allows users to create and host websites directly from their GitHub repositories.

With GitHub Pages, you can turn your GitHub repositories into websites by simply adding HTML, CSS, and JavaScript files, or even Markdown files for static site generation using tools like Jekyll.

These websites are then served directly from GitHub's servers, making it easy to publish documentation, showcase projects, or create personal blogs without the need for external hosting services. GitHub Pages also supports custom domain names, HTTPS encryption, and automatic builds for Jekyll sites, making it a convenient and versatile platform for hosting static websites.

Note: *GitHub pages are free for public repositories only, for private repository you must pay for it.*

45. What is Pull Request in GitHub?

Ans -> A Pull Request in GitHub is a way for users to propose changes to a repository hosted on GitHub. It allows contributors to suggest modifications, additions, or deletions to the codebase, documentation, or any other files within a repository.

When someone creates a Pull Request (often abbreviated as PR), they're essentially asking the repository maintainer to review and potentially merge their proposed changes into the main branch of the repository.

Pull Requests provide a structured way for collaboration and code review within a project, allowing contributors to discuss, comment, and iterate on changes before they are merged. They are a fundamental part of the open-source development process and are commonly used in both small and large-scale projects to maintain code quality and facilitate collaboration among contributors.

46. What is the Workflow for a PR?

Ans ->

- Do some work locally on a feature branch
- Push up the feature branch to GitHub
- Open a pull request using the feature branch just pushed up to GitHub
- Wait for the PR to be approved and merged. Start a discussion on PR. This part depends on the team structure.

47. What does git merge --no-ff <branch_name> this command do?

Ans -> The git merge --no-ff <branch_name> command in Git ensures that when you merge changes from <branch_name> into your current branch, a merge commit is always created, even if Git could have merged the changes without creating a separate commit. This helps maintain a clear history of changes by recording when branches are merged, which can be useful for tracking project progress and understanding the development timeline.

48. What are branch protection rules in GitHub?

Ans -> Branch protection rules in GitHub are settings that can be applied to branches within a repository to enforce certain restrictions and prevent accidental changes or unauthorized modifications.

These rules can include requirements such as *requiring pull request reviewers before merging*, *ensuring that all tests pass*, and *preventing force pushes* or *branch deletions*. By configuring branch protection rules, repository administrators can enhance the stability, security, and quality of the codebase by enforcing best practices for collaboration and code review.

49. What is Forking in GitHub?

Ans -> Forking in GitHub is the process of creating a personal copy of someone else's repository. When you fork a repository on GitHub, you're essentially making a duplicate of that repository in your own GitHub account. This copy includes all the files, commit history, and branches from the original repository.

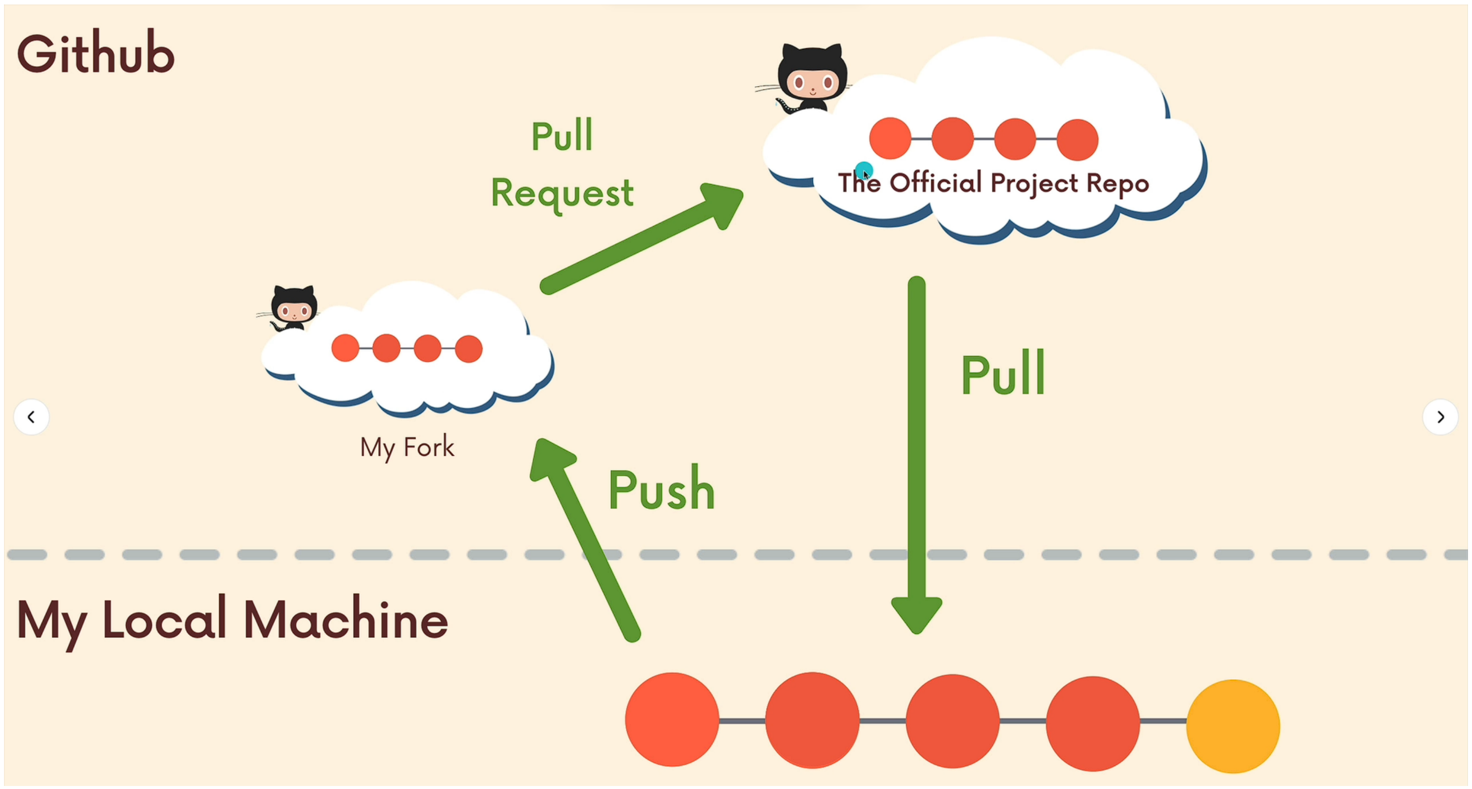
Forking allows you to freely experiment with changes, contribute improvements, or work on new features without affecting the original repository. You can make changes to your forked repository, propose these changes back to the original repository via pull requests, and collaborate with the original repository's maintainers and contributors.

Forking is a fundamental aspect of open-source collaboration on GitHub, enabling decentralized development and community-driven contributions.

50. Demonstrate the workflow of Forking & Cloning.

Ans ->

1. Fork: Click "Fork" on GitHub to create your copy.
2. Clone: Use git close with your Fork's URL to copy it locally.
3. Make Changes: Edit files in your local copy.
4. Commit: Use git add and git commit to save changes.
5. Upstream: Set up a connection to the original repository using git remote add upstream, to pull the latest changes from the original repository.
6. Push: Use git push to send changes to your fork on GitHub
7. Pull Request: Submit a PR on GitHub to propose change to the original repository.



51. What is Rebasing in Git?

Ans -> Rebasing is a Git operation used to integrate changes from one branch onto another by rewriting the commit history.

When you rebase a branch onto another branch, Git:

- Identifies the common ancestor commit of the two branches.
- Temporarily removes the commits from the branch begin rebased.
- Applies each commit from the branch being rebased onto the tip of the other branch.
- Re-establishes the commits with new commit IDs.

It should be used carefully, especially when collaborating with others, as it rewrites the commit history.

Note: *we use Rebase in git for two things:*

- as an alternative to merging
- as a cleanup tool

52. What is the difference between merge and rebase in git?

Ans ->

Merge: When you merge branches in Git, it creates a new "merge commit" that combines the changes from both branches. This preserves the commit history of both branches, showing the merge point and the divergent paths each branch took.

Rebase: Rebase, on the other hand, takes the commits from one branch and places them on top of the commits in another branch. This results in a linear commit history without any merge commits, making the history cleaner and easier to understand. However, it effectively rewrites the commit history of the branch being rebased, which can cause issues if the branch has been shared with others.

In simpler terms, Merge creates a new commit to combine changes from two branches, keeping their original history intact. Rebase moves your changes onto the tip of another branch, creating a linear history but rewriting the commit history of your branch.

53. How to perform Rebase in git?

Ans ->

<u>To rebase in Git, follow these steps:</u>

- Switch the Branch: First, switch to the branch you want to rebase. For example:
  git switch <branch_name>
- Choose the Base Branch: Decide which branch you want to rebase onto. For example, if you want to rebase onto the "master" branch: git rebase master
- Resolve Conflicts (if any): If Git encounters any conflicts during the rebase process, it will pause and ask you to resolve them. You'll need to edit the conflicted files, then use git add to stage the resolved changes.
- Continue Rebase: After resolving conflicts, continue the rebase process with the command: git rebase –continue

54. What is Interactive Rebase in git?

Ans ->

Running git rebase with the -i option (git rebase -i HEAD~<no.> or git rebase -i <commit_hash>) will enter the interactive mode, and Git will open a text editor with a list of commits. Each commit will have options next to it, such as "pick", "reword", "edit", "squash", or "drop". You can change these options to modify the commit history:

- pick: Keep the commit as is.
- reword: Change the commit message.
- edit: Pause the rebase process to make changes to the commit.

- squash: Combine the commit with the previous one and allows you to change the commit message.

- fixup: Combine the commit with the previous one, but you cannot change the commit message here.

- drop: Remove the commit from the history.

55. What are Git Tags?

Ans -> Git tags are labels used to mark specific points in a Git repository's history, typically to indicate important milestones such as releases or version numbers (e.g., v4.1.0, v4.1.1 etc.). They are immutable pointers to specific commits, allowing you to easily reference and identify those commits in the future. Tags are often used to denote stable versions of software, making it simple to retrieve a particular version at any time. Unlike branches, tags do not move when new commits are made; they serve as fixed reference points in the commit history.

In other words, Git tags are like sticky notes you put on important snapshots of your project's history, such as major releases or milestones. They help you easily find and remember those specific points in time, making it convenient to refer back to them later.

git tag -> This command will list done all the available tags.

git checkout <tag_name> -> This command is used to switch to the specified tag, useful for reviewing historical versions or working from stable releases. However, it puts you in a "detached HEAD" state, meaning changes won't belong to any branch unless you create or switch back to one.

git diff <tag_name1> <tag_name2> -> This command shows changes between two tag versions, useful for comparing different releases.

git tag -d <tag_name> -> This command is used to delete a particular tag from our git repository.

56. What are the different types of Tags available in Git?

SAns -> There are two types of Git Tags we can use: lightweight and annotated tags.

lightweight tags: These tags are just a name/label that points to a particular commit.

To create a lightweight tag, use git tag <tag_name>.

By default this creates a lightweight tag at the current commit to which the HEAD is pointing in your repository without any additional metadata. Lightweight tags are essentially just pointers to specific commits in Git's history and don't store any extra information such as tagger name, email, or message.

annotated tags: These tags store extra meta data including the author's name and email, the data, and a tagging message (like a commit message).

git tag -a <tag_name> -> this command is used to create a new annotated tag. Git will then open your default text editor a prompt you for additional information

Similar to git commit, we can also use the -m option to pass a message directly and forgo the opening of the text editor. For example: git tag -a <tag_name> -m "<message>"

This command creates an annotated tag at the current commit to which the HEAD is pointing in your repository with the specified message. Annotated tags store extra information such as the tagger's name, email, the date the tag was created, and the annotated message. Annotated tags are often preferred for release versions or important milestones as they provide more context and information compared to lightweight tags.

git show <tag_name> -> This command is use to display the contents of the annotated tag.

57. What is Semantic Versioning in Git?

Ans -> Semantic Versioning, often abbreviated as SemVer, is a versioning scheme used for software development to communicate the nature of changes in a release. It consists of three numbers separated by dots: MAJOR.MINOR.PATCH (e.g., 2.4.1)

- MAJOR Release: Major releases signify significant changes that are no longer backwards compatible. Features may be removed or changed substantially.

- MINOR Release: Minor releases signify that new features or functionality have been added, but the project is still backwards compatible. No breaking changes. The new functionality is optional and should not force users to rewrite their own code.

- PATCH Release: Patch releases normally do not contain new features or significant changes. They typically signify bug fixes and other changes that do not impact how the code is used.

Semantic Versioning helps developers and users understand changes between versions, managing compatibility and expectations for updates.

58. What does git tag -l command do?

Ans -> We can search for tags that match a particular pattern by using git tag -l and then passing in a wildcard pattern. For example:

The git tag -l "*beta*" command lists all tags in the Git repository that contain the word "beta" in their name. The -l option is shorthand for --list, indicating that you want to list tags, and *beta* is a pattern that matches any tag containing the word "beta".

In simpler terms, it helps you find and list all tags that have "beta" in their name, making it useful for searching for specific types of tags in your repository.

59. How to tag previous commits in git?

Ans ->

git tag <tag_name> <commit_hash> -> This command is used to create a lightweight tag for a particular commit.

git tag -a <tag_name> <commit_hash> -> This command is used to create an annotated tag for a particular commit.

60. How to move tags from one commit to another commit in git?

Ans -> git tag <tag_name> <new_commit_hash> -f -> This command will move the existing tag from the present commit to another commit which we've mentioned in the command.

Here the '-f' is denoted as forced. Typically, we cannot re-use the same tag name, so if we want to some other commit to be tagged as something which already exists, then we have to move that tag name forcefully from previous commit to the commit we want.

61. How to push tags in git?

Ans -> By default. The git push command doesn't transfer tags to remote servers.

git push <remote_name> --tags -> This command is used to push all the available tags from your local machine to remote server.

git push <remote_name> <tag_name> -> This command is used to push a particular tag from your local machine to remote server.

62. What is the purpose of config file inside .git/ folder?

Ans -> The config file inside the .git/ folder serves as a repository-specific configuration file in Git, it's like a settings menu for your Git repository.

It allows us to customize settings such as *user information, default branch names, remote repository URLs, and behavior-related options*. Additionally, we can configure hooks to run custom scripts at specific point during Git operations, providing flexibility and control over the repository's behavior.

Overall, the config file enables us to tailor the setting and behavior of the Git repository to meet the requirements of the project and its collaborators.

63. What is Git cherry-picking?

Ans -> Git cherry-picking is a way to copy a specific commit from one branch and apply it onto another branch. It allows you to pick and choose individual commits to bring over to another branch, rather than merging entire branches together. This can be useful when you only want to include certain changes from one branch into another, without bringing over all the changes from the source branch.

Cherry-picking essentially lets you select specific commit to apply to your current branch, helping you manage and incorporate changes more selectively.

git cherry-pick <commit_hash_from_any_branch> -> This command will bring the specified commit changes to your current branch and commit them right away.
git cherry-pick <commit_hash_from_any_branch> -n -> This command will bring the specified commit changes to your current branch, but it will not commit those changes right away. Here the "-n" refers to no commit.

64. What is the purpose of refs/ folder inside the .git/ directory?

Ans -> The refs/ folder inside the .git/ directory serves as a repository's reference storage in Git. It organizes references to *commits, branches, tags, and other objects* within the repository.

The heads/ subfolder stores references to local branches heads, tags/ contains references to tags, and remotes/ holds references to remote branches.

These structured folders enable Git to efficiently track and access information about branches tags, and other objects, facilitating version control operations like branching, merging, and tagging within the repository.

65. What is the purpose of HEAD file inside .git/directory?

Ans -> The HEAD file inside the .git/ directory serves as a pointer to the currently checked out branch or commit in a Git repository. It indicates the branch or specific commit that your working directory is currently based on. If you're on detached Head state (i.e., not on any branch), HEAD points directly to a specific commit.

Git uses the information in the HEAD file to determine the context of your working directory and to track the changes made in relation to the current state of the repository.

66. What is the purpose of the objects/ folder inside the .git/ directory?

Ans -> The objects/ folder inside the .git/ directory is where Git stores all the compressed and de-duplicated content (*blobs, trees, commits, and tags*) that make up the history and contents of a Git repository. Each object in Git, such as files, directories, and commits, is assigned a unique SHA-1 hash based on its content. These objects are then stored in subfolders inside the objects/ directory, with each subfolder named after the first two characters of the object's hash.

The purpose of the objects/ folder is to efficiently store and manage the history and contents of a Git repository, ensuring data integrity and facilitating operations such as branching, merging, and version control. Git uses these objects to reconstruct the state of the repository at any given point in time and to track changes made to files and directories over the course of the project's history.

67. Which hashing function does Git use and how long hexadecimal numbers does it generates?

Ans -> Git currently uses a hashing function called SHA-1 (though this is set to change eventually).
SHA-1 always generates 40-digit hexadecimal numbers.

68. How Git stores data?

Ans -> Git is a key-value data store. We can insert any kind of content into a Git repository, and Git will hand us back a unique key we can later use to retrieve that content.

These keys that we get back are SHA-1 checksums.

69. What does git hash-object command do?

Ans ->

git hash-object <file_name> -> This command computes the SHA-1 hash of a file's contents without actually staging it in the index. It's useful for checking the integrity of files or creating Git objects directly.

echo "hello" | git hash-object --stdin -> This command will compute the SHA-1 hash of the string inside the quotation (e.g., "hello") without storing it in the repository. It's a way to generate a Git object hash for data passed via stdin.

echo "hello" | git hash-object --stdin -w -> Adding -w will not only compute the hash but also write the object into the Git object database, essentially storing it in the repository.

70. How to retrieve data from Hash Object?

Ans ->
git cat-file -p <object_hash> -> This command is used to display the content of a Git object identified by its object hash id. The -p option tell Git to pretty print the contents of the object based on its type.

71. What are the different types of Git Objects available, explain them in short.

Ans -> In Git, there are four main types of objects:

Blob (Binary Large Object): Git blobs are the object type Git uses to store the contents of files in a given repository. Blobs don't even include the filenames of each file or any other data. They just store the contents of a file.

Tree: Trees are Git objects used to store the contents of a directory. Each tree contains pointers that can refer to blobs (file contents) and to other trees (subdirectories).
Each entry in a tree contains the SHA-1 hash of a blob or tree, as well the mode, type, and filename.

Commit: Commit object combine a tree object along with information about the context that led to the current tree. Commits store a reference to parent commit(s), the author, the commiter, and of course the commit message.

Annotated Tags: Contains metadata such as tagger information, tag message, and reference to the commit they tag.