# Interview Questions

1. **What is Git**?
   1. Git is a distributed version control system (VCS) that is widely used in software development to track changes in source code during the development of a project. It was created by Linus Torvalds in 2005 and has since become the de facto standard for version control.
2. **What are key concepts and features of GIT**?
   1. Here are some key concepts and features of Git:
   2. **Version Control System (VCS)**: Git is a version control system that allows multiple developers to collaborate on a project. It keeps track of changes made to files over time, allowing you to revert to previous states, compare changes, and collaborate with others efficiently.
   3. **Distributed System**: Git is a distributed version control system, meaning that each developer has a complete copy of the entire repository, including its history. This allows developers to work offline and makes the system more robust.
   4. **Branching and Merging**: Git provides powerful branching and merging capabilities. Developers can create branches to work on new features or bug fixes independently. Changes made in one branch can be merged back into the main codebase when ready.
   5. **Commit**: A commit in Git represents a snapshot of the project at a specific point in time. Each commit has a unique identifier, a commit message describing the changes made, and references to the changes.
   6. **Repository**: A Git repository is a collection of files and directories, along with the version history and metadata. It can be local (on a developer's machine) or remote (hosted on a server, e.g., GitHub, GitLab, Bitbucket).
   7. **Remote Repository**: Git allows collaboration by enabling developers to work on a local copy of a repository and push their changes to a remote repository. This is often done using hosting services like GitHub, which provide a central location for collaboration.
   8. **Pull**: Pulling in Git refers to fetching changes from a remote repository and integrating them into the local repository. It is often used to update the local copy with changes made by others.
   9. **Push**: Pushing in Git involves sending local commits to a remote repository, making them available to other developers.
   10. **Merge Conflict**: A merge conflict occurs when Git is unable to automatically merge changes from different branches. Developers must resolve these conflicts manually.
   11. **GitHub and GitLab**: GitHub and GitLab are popular web-based platforms that host Git repositories. They provide additional collaboration features, such as issue tracking, pull requests, and code reviews.
       1. Git is an essential tool in modern software development, enabling efficient collaboration, version control, and project management. It is used by individual developers, small teams, and large organizations for a wide range of projects.
3. **What are the fundamental concepts of Git**?
   1. In the context of Git and GitHub, the terms "Working Directory," "Staging Area," "Local Repository," and "Remote Repository" are fundamental concepts that represent different stages in the Git workflow.
   2. **Working Directory**:
      1. The working directory is the directory on your local machine where you are currently working on your project. It contains the actual files of your project.
      2. These files may be in various states—some might be modified, others unmodified, and some may be newly created.
      3. Changes made to files in the working directory are not automatically tracked by Git until you explicitly tell Git to do so.
   3. **Staging Area (Index)**:
      1. The staging area, also known as the index, is an intermediate area where you can prepare and organize changes before committing them to the version control system.
      2. After making changes in the working directory, you use the git add command to move these changes to the staging area. This process is often referred to as staging changes.
      3. The staging area allows you to selectively choose which changes to include in the next commit.
   4. **Local Repository**:
      1. The local repository is a hidden directory within your project that contains the complete history and snapshots of your project at various points in time.
      2. When you commit changes using the git commit command, the changes in the staging area are saved as a new commit in the local repository.
      3. The local repository is where Git stores the full history of your project, and you can use Git commands to navigate through this history.
   5. **Remote Repository**:
      1. The remote repository is a repository that is hosted on a server, often on platforms like GitHub, GitLab, or Bitbucket.

|  |  |  |  |
| --- | --- | --- | --- |
| **Working Directory** | **Staging Area** | **Local Repository** | **Remote Repository** |
| Write your code here and save files. Git init to make normal folder as a git folder working directory | Before sending the code/files into your local repository this staging area will hold the code files | In Git, a local repository refers to the version of your project that is stored on your local machine. This local repository contains all the files, history, and branches of your project, allowing you to work on it, make changes, and commit those changes without needing to connect to a remote server. | git remote -v  The git remote -v command displays the URLs of the remote repositories associated with your local Git repository. The output will typically show both the fetch and push URLs for each remote.  git remote: This command deals with the remote repositories. The -v option stands for "verbose" and displays the details in a more descriptive format.  When you run git remote -v, you might get an output that looks something like this:  origin https://github.com/username/repository.git (fetch)  origin https://github.com/username/repository.git (push)  \*if no remote is already configured, this command will return empty/nothing.  In this example:  **origin** is the name of the remote repository. By convention, the first remote added to a git repository is named origin, but you can have multiple remotes with different names.  The URL https://github.com/username/repository.git is the address of the remote repository. The (fetch) and (push) annotations indicate the fetch and push URLs, respectively. |
| Check if the status of git using **git status**. To send your code files (untracked files) to staging area use **git add .** | You can send the files from staging area to Local repository by using the command  **git commit -m** “<your message comment>” | **.git** Directory: At the root of your project directory, you'll find a hidden .git directory. This directory is where Git stores all the information about the project's history and metadata. | **git remote add origin** [**https://github.com/praveenkumarilla459/git-tuts.git**](https://github.com/praveenkumarilla459/git-tuts.git)  With this command we are setting the git remote to a GitHub repo named git-tuts.  After the above command execution, we can see the below when entered git remote -v: git remote -v  origin https://github.com/praveenkumarilla459/git-tuts.git (fetch)  origin https://github.com/praveenkumarilla459/git-tuts.git (push) |
| From working directory to staging area you can send files, one by one or at a time **git add <filename> git add .** | From staging area to local repository, you can only send all files like a batch of files at a time but not single file at a time. We can also bring back the files from staging area to working directory using the command git rm –cached <filename> **git rm –cached <filename>**  **git rm --cached example.txt** | **Object Database**: Within the .git directory, there are subdirectories that contain objects. These objects represent the various commits, trees (directory structures), and blobs (file contents) in your repository.  **HEAD**: This is a pointer to the currently checked-out commit in your repository. It's essentially where you are in your project's history | The command **git push -u origin main** is commonly used when setting up tracking for a branch in a local repository and pushing it to a remote repository. Let's break down the command step by step:  **git push**: This is the Git command to push commits from your local repository to a remote repository.  **-u**: This option stands for "upstream." When you use **-u**, you're setting the upstream branch for the current local branch. Once set, Git remembers the relationship between the local and remote branches. After setting the upstream, in the future, you can simply use git push or git pull without specifying the remote and branch name, as Git knows where to push or pull changes from.  **origin**: This is the default name Git gives to the remote repository from which you cloned your local repository or to which you've added as a remote using the **git remote add** command. In many repositories cloned from platforms like GitHub, the main remote repository is named origin.  **main**: This refers to the name of the branch you're pushing to the remote repository. In the context of this command, you're pushing the local branch named main to the main branch of the remote repository.  Putting it all together, the command **git push -u origin** main does the following:  Pushes the commits from your local main branch to the main branch of the origin remote repository.  Sets up the local main branch to track the main branch of the origin remote repository. This means in future git push or git pull commands (without specifying branch names), Git knows that you're referring to the origin remote's main branch.  After using this command, you can simply use git push and git pull in the main branch without specifying origin or main, thanks to the upstream setting established by -u. |
|  |  | **Branches**: Git allows you to have multiple branches in a repository, and each branch can represent a different line of development. Branches in the local repository are lightweight, meaning they're just pointers to specific commits  **Commits**: Each time you make a change and commit it in Git, a new commit object is created in the local repository. Commits store metadata like the author, date, and a pointer to the parent commit, allowing Git to track the history of changes in the project | **Instead of giving git remote add origin https://github.com/<myusername/<myreponame>.git if I give git remote add salar https://github.com/<mysuername>/<myreponame>.git**  **my question is can we change origin to someother thing accoridng to our project requirement if needed?**  Yes, absolutely! The names you give to remotes (like origin, salar, etc.) are merely aliases or labels within your local repository. They don't affect the functionality or behavior of Git itself. These names are used as convenient shorthand to refer to specific remote URLs.  Here's a breakdown:  **Remote Aliases**: The names you provide after git remote add are just aliases for the remote repository URLs. This means that you can use salar in place of origin for any Git command.  **Functionality**: Once you've added a remote with a specific name (like salar), you can push, pull, fetch, etc., using that name just as you would with origin.  For instance, git push salar main would push the main branch to the main branch on the remote repository associated with the salar alias.  **Multiple Remotes**: In fact, many repositories often have multiple remotes. For example, you might have one for your primary project repository named origin, another for a forked repository or a backup named backup, and yet another for a collaborator's repository named after them, like salar.  **Default Names**: While origin is a common default name for the primary remote repository, it's just a convention. You can choose any name that makes sense for your project, your workflow, or your team.  So, to directly answer your question: Yes, you can change origin to some other name (like salar or anything else) according to your project requirements. |

1. **What is git clone**?
   1. The ‘git clone’ command is used to create a copy of a remote Git repository on your local machine. This command initializes a new Git repository and fetches the entire history and files from the specified remote repository. The cloned repository on your local machine will have the same commit history and file structure as the original repository.
   2. The basic syntax for ‘**git clone**’ is as follows:
      1. git clone <repository\_url>
   3. When you clone a repository from a remote server, you create a local copy of the remote repository on your machine. You can also add a remote repository to an existing local repository using the git remote command.
      1. Example: git clone <https://github.com/example/repository.git>
   4. Here's a breakdown of what happens when you run git clone:
   5. **Initialization**: Git initializes a new repository in the specified destination directory on your local machine.
   6. **Remote Connection**: Git establishes a connection to the remote repository specified by the <repository\_url>.
   7. **Fetch**: Git fetches all the branches, commits, and files from the remote repository to your local machine.
   8. **Default Branch**: The default branch (often named "master" or "main") is checked out in your local repository.
   9. Here's a breakdown of what happens when you run git clone:
   10. After the “**git clone**” command completes, you have a complete copy of the remote repository on your local machine, and you can start working with the code. The connection to the remote repository is preserved, allowing you to fetch updates from the remote repository later.
   11. Additional options can be used with git clone to specify a different destination directory, clone a specific branch, or use other configuration options. For example:
       1. git clone -b branch\_name <repository\_url>
   12. This command clones the specified branch (branch\_name) from the remote repository.
   13. The git clone command is a fundamental step when starting to work on a project, especially when collaborating with others or when setting up a local development environment for an existing project hosted on platforms like GitHub, GitLab, or Bitbucket.
   14. The remote repository is where you can collaborate with other developers. You can push your local commits to the remote repository, and you can also fetch changes made by others.
2. In the context of GitHub:
   1. GitHub:
   2. GitHub is a web-based platform that provides hosting for Git repositories. It allows you to store your repositories remotely and provides additional collaboration features such as pull requests, issues, and project management tools.
   3. Origin:
   4. In Git, "origin" is a default name commonly used to refer to the remote repository from which the local repository was initially cloned. You typically push changes to and pull changes from the origin remote.
   5. In summary, the workflow typically involves making changes in the working directory, staging those changes in the staging area, committing them to the local repository, and then pushing those commits to a remote repository, such as one hosted on GitHub.
3. What is **git init**?
   1. **Git init**:
   2. The git init command is used to initialize a new Git repository. When you run this command in a directory, Git sets up the necessary data structures and files, creating a new repository from scratch.
4. What is ‘**git status**’?
   1. git status is a command used in the Git version control system to display the state of the working directory and the staging area. When you run git status in your terminal or command prompt while inside a Git repository, it provides information about:
   2. **Changes to be committed**: This section shows the changes that are staged (added to the index) and ready to be committed to the repository.
   3. **Changes not staged for commit**: These are the changes in the working directory that have not yet been staged (added to the index). In other words, these are modifications that you have made but haven't yet told Git that you want to include in the next commit.
   4. **Untracked files:** These are files in your working directory that are not tracked by Git. Git is unaware of these files, and they won't be included in commits unless you explicitly add them.
   5. The git status command is useful for understanding the current state of your repository and what changes are pending. It helps you decide what actions to take next, such as staging files for a commit or adding new files to the repository.
5. What is **‘git add’**?
   1. **git add** is a Git command used to add changes in the working directory to the staging area. The staging area, also known as the "index," is a place where changes are organized before they are committed to the repository. By adding changes to the staging area using git add, you are preparing them for the next commit.
   2. **Add Specific Files**: You can add specific files to the staging area by specifying their names: **git add filename.txt**
   3. **Add All Changes**: You can add all changes in the working directory to the staging area using: **git add .**
      1. The . indicates the current directory. This command stages all changes in the current directory and its subdirectories.
   4. **Add All Changes Interactively**: Git provides an interactive mode for adding changes, where you can selectively add changes from specific files or parts of files. To enter interactive mode, use: **git add -i**
      1. This command opens an interactive interface where you can choose which changes to stage.
      2. After using git add to stage changes, you can review the changes using git status to see which files are staged and which files have changes that are not yet staged. Once you have added all the changes you want to the staging area, you can commit them using git commit.
6. What is **git commit**?
   1. **git commit** is a command used in the Git version control system to record the changes made to the repository. When you make changes to your project files in a git repository, you can stage these changes (using git add) and then commit them with a message to describe the changes.
   2. **Staging Changes**: Before committing, you typically stage the changes you want to commit. This is done using the git add command, which adds changes in the working directory to the staging area (also known as the index.
      1. git add filename.txt
      2. This command stages the changes in filename.txt to be committed
   3. **Creating a Commit**: Once you've staged the changes you want to commit, you can create a commit using the git commit command.
      1. git commit -m "Commit message describing the changes"
   4. **Commit Message**: It's good practice to provide a clear and concise commit message that describes what the commit accomplishes. A well-written commit message helps other developers (and your future self) understand the purpose and context of the changes.
   5. **Commit SHA**: After committing, Git generates a unique identifier called a commit SHA (Secure Hash Algorithm) for that commit. This SHA can be used to reference the commit in various Git commands and operations.
   6. **Recording Changes**: When you make a commit, Git records a snapshot of the changes that have been staged. This snapshot includes information about which files were changed, the contents of those files, and the parent commit (the commit from which the changes originated)
   7. In summary, the **git commit** command is essential for saving and documenting changes to your Git repository, allowing you to track the history of your project and collaborate with others effectively.
7. Explain more on **Commit SHA** and how does it look?
   1. The Commit SHA (Secure Hash Algorithm) is a unique identifier generated by Git for each commit. This SHA serves as a unique fingerprint for the commit, ensuring that each commit in a repository has a distinct identifier. The Commit SHA is crucial for referencing, tracking, and working with commits within a Git repository.
   2. **Uniqueness**: Each commit in a repository has a unique Commit SHA. Even if two commits have almost identical changes, they will have different Commit SHAs due to the hash algorithm's properties
   3. **Length**: A Commit SHA typically appears as a 40-character hexadecimal string. For example: aefead2207ef7e2aa5dc81a34aedf0a4c5d19a0a. The length and format remain consistent across Git repositories and implementations.
   4. **Immutable**: Once a commit is created, its SHA does not change. This immutability ensures that references to commits remain valid and consistent throughout the repository's history.
   5. **How Commit SHA is Generated**:
   6. **Content-based**: The Commit SHA is derived from the contents of the commit, including the commit message, author, timestamp, parent commits, and the changes made in the commit.
   7. **Hash Algorithm**: Git uses the SHA-1 cryptographic hash function to generate the Commit SHA. The SHA-1 algorithm processes the commit's content and produces a 160-bit (or 40-character hexadecimal) hash value, which becomes the Commit SHA.
   8. **Importance of Commit SHA**:
   9. **Referencing Commits**: The Commit SHA allows you to reference and work with specific commits, even if they are deep in the repository's history. You can use the Commit SHA in various Git commands, such as **git log**, **git checkout**, and **git reset**, to view, navigate, or modify commits
   10. **Integrity**: The Commit SHA ensures the integrity and authenticity of commits. If any part of a commit is altered (accidentally or maliciously), its SHA will change, signaling a modification or corruption.
   11. **Collaboration**: When collaborating with others, the Commit SHA provides a reliable way to share and discuss specific commits. By referencing Commit SHAs, team members can ensure they are discussing the same commit and avoid confusion
   12. In summary, the Commit SHA is a fundamental aspect of Git that provides a unique and immutable identifier for each commit. It plays a vital role in tracking changes, maintaining repository integrity, and facilitating collaboration within Git repositories
8. Explain **git log** command:
   1. **git log** is a command in Git that displays the commit history of a repository. It provides a chronological list of commits, allowing you to see who made changes, when they were made, and the associated commit messages.
   2. Here's how to use git log and some of its common options:
      1. git log
         1. This command shows the commit history with the following details:
         2. Commit SHA (Unique identifier)
         3. Author name and email
         4. Commit date and time
         5. Commit message
   3. **Display Commit Details**:
      1. To see more detailed information about each commit, including the changes made:
      2. git log -p
      3. The -p option (or --patch) displays the diff of each commit, showing the actual changes made in each commit.
   4. **Limit the Number of Commits**:
      1. To limit the number of commits displayed (e.g., show only the last 3 commits):
         1. **git log -n 3**.
   5. **Display Summary Only:**
      1. If you're only interested in a concise summary of each commit:
      2. **git log –oneline**.
      3. This displays each commit as a single line, showing the Commit SHA and the commit message.
   6. **Graphical Representation**:
      1. To visualize the branching and merging in the commit history:
         1. **git log --graph --oneline –all**
      2. This command provides a compact graphical representation of the commit history, making it easier to understand the branching structure.
   7. **Filter by Author**:
      1. To view commits made by a specific author:
      2. **git log --author="Author Name"**.
   8. **Filter by Date**:
      1. To view commits made within a specific date range:
      2. **git log --since="2023-01-01" --until="2023-12-31"**.
   9. **Search by Commit Message**:
      1. To search for commits with a specific keyword in the commit message:
      2. **git log --grep="keyword"**.