Version Control Interview Questions

(Practice Project)

PW SKILLS

Interview Questions

Easy

Q. What is a version control system (VCS)?
Ans:
A version control system, or VCS, is a software tool that helps manage and track changes to source code or any set of files over time. It allows multiple contributors to work collaboratively on a project, keeping track of modifications, and providing mechanisms to merge changes seamlessly.

Q. What are the advantages of using a VCS?
Ans -
The advantages of using a version control system include

1. Single Source of Truth
2. Full file history
3. Improve Visibility
4. Efficient collaboration
5. Traceability
6. Streamlined Branching and Merging
7. Code experimentation
8. Source code Management
9. Efficient and Productivity

Q. How does Git differ from other version control systems
Ans -
Git differs from other version control systems in several key ways:

1. Distributed Model
2. Branching and Merging
3. Speed
4. Offline Capability
5. Open Source and Cross-Platform

Q. What is a git repository?
Ans:
A Git repository is a storage location that holds a collection of files and their revision history. It includes all the files and directories associated with a project, along with the metadata generated by Git. This repository allows for version control, enabling multiple developers to collaborate on a project and track changes efficiently.

Q. What is the difference between Git and GitHub?
Ans:
Git is a distributed version control system (DVCS), while GitHub is a web-based platform that provides hosting and storage for Git repositories. Git is the underlying technology that manages version control locally, whereas GitHub offers a centralized platform for collaboration, issue tracking, and project management.

Q. What are the advantages of using GIT?
Ans:
Using Git offers several advantages, including:

- **Distributed Development:** Git allows developers to work independently on their local repositories and merge changes seamlessly.
- **Version Control:** Tracks changes, facilitating collaboration and providing a history of modifications.
- **Branching and Merging:** Enables efficient management of code branches, allowing for parallel development and easy integration of changes.
- **Speed:** Since Git has all the history saved on your local computer, it works faster because you can make changes and save them without needing to be connected to the internet.

Q. What is the difference between Git and SVN?
Ans -
Git and SVN are two different version control systems used for managing changes to code and other digital content. The main differences between Git and SVN are:

1. **Distributed vs Centralized:** Git is a distributed version control system, which means that every user has a full copy of the repository and can work on it independently. SVN, on the other hand, is a centralized system where all the code is stored on a single server.

2. **Speed:** Git is known for its speed and efficiency, making it a popular choice for high-traffic projects. SVN can be slow and cumbersome, especially when compared to Git.

3. **Branching:** Git allows developers to create and switch between different branches in order to add features or fix bugs without disrupting the main development branch and the production codebase. SVN does not have this feature.

4. **Repository Structure:** Git stores the full history of all of its branches and tags within the .git directory. SVN stores the history in a central server.

5. **Subprojects:** Git allows for submodules, which are projects that are developed and managed outside of the main project. SVN does not have this feature.

Q. What is the function of 'git config'? how to config user details
Ans:
The git config command is used to configure Git settings either globally, per user, or per repository. It allows users to set parameters like user name, email, preferred editor, and other options that affect the behavior of Git.

Q. Explain what does the git clone do?
Ans -
Git clone is a command used to create a local copy of an existing remote repository. When you clone a repository, you get an exact copy of all the files, branches, and commit history from the original repository.

```
git clone [alias][remote-url]
```

Q. How to create a new git repository in an existing project.
Ans:
To create a new Git repository in an existing project, you can use the following commands:

```
git init
```

This initializes a new Git repository in the current directory, marking it as a place to track changes and revisions.

Q. What is the git push command?
Ans:
The git push command is used to upload local repository content to a remote repository. It sends the committed changes from your local branch to the corresponding branch on the remote repository.

Q. What is the git pull command?
Ans:
The git pull command is used to fetch changes from a remote repository and merge them into the current branch. It combines the git fetch and git merge operations into a single command.

Q. What is the purpose of a .gitignore file?
The purpose of a .gitignore file is to specify files or directories that should be ignored by Git. This file is used to prevent Git from tracking certain files or directories that are not part of the source code, such as binary files, temporary files, or files generated during the development process. The .gitignore file is typically placed in the root directory of a project and lists patterns for files or directories to ignore.
Example of the .gitignore file

```
# dependencies
/node_modules
/.pnp
.pnp.js
# testing
/coverage
# production
/build
```

Q. Explain some Git commands to get started with version control
Ans -

```
// 1. git init - initialize a new Git repository in an
existing project.
git init

// 2. git clone - make a copy of a remote Git repository on
a local machine
git clone

// 3. git status - check and display the current status of
the project
git status

// 4. git add - add new files to begin tracking
git add [file]
// 5. git commit - used to save staged changes with unique
hash
git commit -m [descriptive commit message]

// 6. git push - save all commit chanes to remote
repository
git push [alias][branch] // alias can be origin or remote
```

Q. Explain the difference between a working directory, staging area, and repository in git
Ans -
Working Directory: The working directory is the directory on your local machine where you work on your project files. It contains all the files and folders in your project, including those that are tracked by Git and those that are not. This is where you make changes, create new files, or delete files. The working directory represents the current state of your project files, which may not yet be committed to the Git repository.

Staging Area (Index or Cache): The staging area, also known as the index or cache, is an intermediate area that stores changes to tracked files before they are committed to the repository. It serves as a way to organize and review the changes you have made to your project before creating a new commit.

Medium

Q. How do you find a list of files that have changed in a particular commit?

Ans:
There are two main ways to find a list of files that have changed in a particular commit in Git:

1. Using the **git log** command
2. Using **git diff-tree** command

Q. What is the difference between git pull and git fetch
Ans -

```
// git fetch - retreive updates from remote repo without
merging
git fetch

// git pull - retrieve and merge update from remote repo
git pull
```

Q. What does 'git commit -am' command?
Ans:
The git commit -am command is a shorthand way to create a new commit in the Git version control system. It combines the git commit command with the -a and -m flags

Q. What is the 'git stash' command?
Ans:
The git stash command is a temporary storage area for your tracked changes. It is used to save your current work in progress and revert your working directory to its last commit state. This can be useful if you need to switch branches or work on something else and want to come back to your changes later

Q. Why do we need git stash
Stashing refers to temporarily saving your work in progress, including both staged and unstaged changes, on a stack of unfinished changes that you can reapply later.

Stash can be done with the following commands -

```
// save modified and staged changes
git stash

// list all stash file changes
git stash list

// get working from the top of the stash
git stash pop

// clear stash
git stash drop
```

Q. Explain the use of the following commands

```
git branch --merged
git branch --no-merged
```

Ans -

```
git branch --merged
- This command lists branches that have been merged into
the current branch. It helps identify branches that have
already been integrated into the current branch, making it
safe to delete them. When you run git branch --merged, Git
displays a list of branches that have been successfully
merged into the current branch.

git branch --no-merged
- This command lists branches that have not been merged
into the current branch. It helps identify branches that
contain work that has not yet been incorporated into the
current branch. When you run git branch --no-merged, Git
displays a list of branches that have changes not yet
merged into the current branch.
```

Q. Why do we need branching in GIT? Define ways to create a branch.
Ans:
Branching is an essential feature in Git, enabling developers to manage different versions of their code without affecting the main codebase. It allows for parallel development, experimentation, and collaboration without the risk of disrupting the current stable version of the code.

Ways to create a branch -
- Using the git checkout -b command
- Using the git branch command

Q. How can you revert a bad commit that is already pushed?
Ans -
To revert a bad commit that is already pushed, you can use the git revert command. This command creates a new commit that undoes the changes introduced by the bad commit. Here's how you can do it:

```
git revert [commit_hash] // revert and then push
git push orign [branch-name]
```

Q. What is the command to list all branches?
Ans:

To list all branches of Git use the **git branch**

```
git branch
```

Q. What is the difference between 'git remote' and 'git clone'?
Ans:
git remote -
The git remote command is used to add, remove, or modify remote repositories associated with a local Git repository. It allows you to establish connections to remote repositories and manage their URLs and names.

git clone -
The git clone command is used to create a local copy of a remote repository. It fetches the entire contents of the remote repository and creates a local copy with the same structure and history.

Q. What is the difference between git rebase and git merge?
Ans:

Feature	Git Merge	Git Rebase
Commit History	Creates a new merge commit	Rewrites commit history
Branch Structure	Preserves branch structure	Flatten branch structure
Use Cases	Suitable for combining work from multiple contributors	Suitable for integrating changes from a single source
Collaboration	More suitable for collaborative projects	More suitable for individual or private projects

Q. What is the difference between git fetch and git pull?
Ans:

Feature	Git Fetch	Git Pull
Data Acquisition	Fetches new data from the remote repository	Fetches and merges new data from the remote repository
Working Directory	Does not affect the working directory	Merges changes into the working directory
Local Branch	Updates tracking information	Merges changes into the local branch
Use Case	Useful for updating local knowledge of remote changes	Suitable for integrating the latest changes from the remote repository

Q. How do you revert a commit?
Ans:
There are two main ways to revert a commit in Git
- Using the git reset command
- Using the git revert command

Q. Mention the difference between revert and reset git command
Ans -
git revert: The git revert command creates a new commit that undoes the changes made by a specific commit. It effectively adds a new commit that negates the changes introduced by the commit being reverted. This approach keeps a clear record of the changes made and avoids altering the commit history. git revert is a safe option for undoing changes that have already been pushed to a remote repository.

git reset: The git reset command is used to reset the current branch to a specific commit, effectively moving the branch pointer to a different commit. It can be used to undo changes made in the working directory, staging area, or commit history. There are different modes of git reset (--soft, --mixed, --hard) that determine how the reset affects the working directory, staging area, and commit history. git reset can be a more powerful but riskier command compared to git revert, as it can modify the commit history and potentially lead to data loss. It is typically used for local operations and should be used with caution, especially when dealing with published commits.

Q. Could you explain the distinction between staging and stashing in Git?
Ans -
Staging refers to preparing changes in your working directory to be included in the next commit. When you make changes to files in your Git repository, these changes are initially considered to be in the "unstaged" state. Staging involves selectively choosing which changes you want to include in the next commit.
It can be done with the git add command

```
// to add specific file
git add [file]
// to all change files
git add .
```

```
// save modified and staged changes
git stash

// list all stash file changes
git stash list

// get working from the top of the stash
git stash pop

// clear stash
git stash drop
```

Hard

Q. How can you find a commit that broke something after a merge conflict
Ans -
To find a commit that broke something after a merge conflict, you can use the git bisect command. Here's how you can do it:

```
bisecting - Identifies the commit that introduced a bug
using a binary search approach.
git bisect start    // starts a bisect session
git bisect bad      // marking the current HEAD as bad
git bisect good [commit-hash] // marks as know good commit
```

Q. What is semantic versioning in context of Git tagging?
Ans -
Semantic versioning (SemVer) is a set of rules and guidelines for assigning version numbers to software releases. In the context of Git tagging, semantic versioning can be used to tag commits with meaningful version numbers that convey information about the changes in each release.

The semantic versioning specification defines a version number in the format MAJOR.MINOR.PATCH, where:

- MAJOR version is incremented when incompatible API changes are made.
- MINOR version is incremented when new functionality is added in a backwards-compatible manner.
- PATCH version is incremented when backwards-compatible bug fixes are made.

When using semantic versioning with Git tags, you would create a tag for each release with the corresponding version number. For example:

```
git tag v1.2.3
```

Q.What is HEAD in Git?
Ans -
In Git, HEAD is a reference that points to the currently checked-out branch or commit in the repository. It is a special pointer that indicates the commit you are currently viewing or working on. HEAD is crucial for tracking changes and managing multiple versions in a Git repository. When you switch branches or check out a specific commit, HEAD is updated to reflect the new state of the repository.

Q. How can you perform Squash commit in Git?
Ans -
To perform a squash commit in Git, you can use the git squash command -

- Select the commits to squash: Choose the commits you want to squash by selecting them in the Git GUI or by using the git log command to identify the commit hashes.
- Run git squash: Use the git squash command to squash the selected commits into a single commit.
- For example:

```
git squash <commit_hash_1> <commit_hash_2> .
```

- Commit the squashed commit: Once the squash operation is complete, you'll be prompted to commit the squashed changes. Provide a meaningful commit message to describe the changes.
- Push the changes: Push the squashed commit to the remote repository to update the shared history.

Q. What is a conflict in git and how can it be resolved?
Ans:
In Git, a conflict occurs when changes are made to the same part of a file by different branches or contributors, and Git cannot automatically merge these changes. This typically happens during a merge or rebase operation. To resolve a conflict, follow these steps:

1. **Identify the conflicted files:** Git will mark the conflicted files, usually with "<<<<<<<", "=======", and ">>>>>>>". Open these files in your code editor.
2. **Manually resolve the conflict:** Within the conflicted file, locate the conflicting sections and modify them to represent the desired final state. Remove the conflict markers and any unnecessary code introduced by Git.
3. **Mark as resolved:** After manually resolving the conflict, mark the file as resolved by staging it with git add [filename].
4. **Complete the merge or rebase:** Continue with the merge or rebase process by running git merge --continue or git rebase --continue.
5. **Commit the changes:** Finally, commit the resolved changes with the git commit.

Q. What is HEAD in Git, and how many HEADs can be created in a repository?
Ans:
In Git, the term "**HEAD**" refers to a special pointer that points to the latest commit in the currently checked-out branch. It essentially represents the snapshot of your working directory. You can think of it as the "current commit" or the "tip of the branch." There is only one HEAD per repository.

However, it's important to note that you can have multiple branches in a Git repository, each with its own HEAD. When you switch between branches using git checkout or git switch, the HEAD pointer is updated to point to the latest commit in the newly checked-out branch.

Q. What is Git reflog ? how to recover a deleted branch using it reflog? Mention some git reflog sub command.
Ans -
Git reflog is a reference log that records updates to the tips of branches in a Git repository. It stores a history of all the changes made to the repository, including commits, merges, and other updates. The reflog is a useful tool for recovering lost commits, branches, or changes that may have been accidentally deleted or lost.

To recover a deleted branch using Git reflog, you can follow these steps:

- Identify the commit hash of the deleted branch: Use the reflog to find the commit hash of the deleted branch. You can do this by running:

```
git reflog
```

- Locate the commit hash corresponding to the deleted branch: Look for the entry in the reflog that corresponds to the deleted branch. The reflog will show a list of recent actions, including the deletion of the branch.
- Create a new branch at the commit hash: Once you have identified the commit hash of the deleted branch, you can create a new branch at that commit using:

```
git checkout -b <new_branch_name> <commit_hash>
```

- Replace <new_branch_name> with the name of the new branch you want to create and <commit_hash> with the commit hash of the deleted branch.
- Recover the deleted branch: By creating a new branch at the commit hash where the deleted branch was last seen, you effectively recover the deleted branch with its commit history intact.

Q. What is the regular way for branching in GIT?
Ans:

The regular way for branching in Git involves the following steps:

1. **Create a new branch:** Use the command git branch [branch_name] to create a new branch. For example, git branch feature-branch.
2. **Switch to the new branch:** Use git checkout [branch_name] or git switch [branch_name] to switch to the newly created branch. Alternatively, you can combine the creation and switching with git checkout -b [branch_name] or git switch -c [branch_name].
3. **Make changes:** Work on your code and make the necessary changes in the branch.
4. **Stage and commit changes:** Use git add to stage changes and git commit to commit them to the branch.
5. **Switch between branches:** You can switch between branches using git checkout or git switch. Ensure that you commit or stash changes before switching branches if you have uncommitted changes.
6. **Merge branches:** Once you're done with the changes in a branch, you can merge it back into the main branch or another target branch using git merge.

Q. Explain the git tag with an example
Ans -
Git tags are a way to mark specific points in the history of a Git repository. They are typically used to label important milestones or releases, such as a version, release, or major project update

```
git tag [tag-name] [commit-hash]
git tag -1  // list all tags in your repo
```

Q. What is cherry-pick in Git?
Ans:
In Git, "cherry-picking" refers to the act of selecting and applying a specific commit from one branch to another. This allows you to choose individual changes and apply them to a different branch, rather than merging an entire branch.

Here's how you can use git cherry-pick:

1. **Identify the commit:** First, you need to identify the commit you want to cherry-pick. You can find the commit hash by using commands like git log or through a Git graphical interface.

2. **Switch to the target branch:** Move to the branch where you want to apply the selected commit using git checkout or git switch.

3. **Cherry-pick the commit:** Use the git cherry-pick command followed by the commit hash you want to apply. For example:

```
git cherry-pick [commit-hash]
```

4. **Resolve conflicts (if any):** If there are conflicts during the cherry-pick operation, Git will pause and allow you to resolve them manually. After resolving conflicts, use git add to stage the changes and then continue the cherry-pick with git cherry-pick --continue.

5. **Commit the changes:** Once the cherry-pick is complete, commit the changes with the git commit.

Q. Can you explain the Gitflow workflow?
Ans:
Gitflow is a branching model for Git that provides a set of guidelines for managing and organizing branches in a Git repository. It was introduced by Vincent Driessen and is designed to facilitate collaboration among developers, especially in projects with regular releases. The Gitflow workflow defines specific branch types and their purposes.

Here are the main branches in the Gitflow workflow:

Master Branch (master): Represents the official released code.
Develop Branch (develop): Serves as the integration branch for features.

Feature Branches (feature/feature-name): Created for developing new features or making substantial changes.

Release Branches (release/version-number): Created when the develop branch has accumulated enough features for a release.

Hotfix Branches (hotfix/version-number): Created to fix critical issues in the production code.

The typical sequence of actions in the Gitflow workflow is as follows:

- Developers create feature branches from the development branch.
- Completed features are merged back into the development branch.
- When the develop branch has accumulated enough features for a release, a release branch is created.
- The release branch undergoes final testing and is then merged into both master and develop.
- If issues are discovered in the production code, hotfix branches are created from the master branch to address them.
- Hotfixes are merged back into both master and develop.
- This workflow helps maintain a structured and organized development process, especially in projects with frequent releases and multiple contributors. It ensures that stable code is always available in the master branch while ongoing development and testing occur on the develop branch.

Q. Can you explain the GitHub workflow?
Ans:
The GitHub workflow is a simplified and flexible Git workflow that leverages the features provided by the GitHub platform. It is commonly used for collaborative software development and integrates seamlessly with GitHub's pull request and code review capabilities. The GitHub workflow typically includes the following steps:

1. Create a Fork: Developers start by forking the original repository on GitHub. This creates a copy of the repository under their GitHub account.

2. Clone the Repository: Developers clone their forked repository to their local machine using the git clone command.

```
git clone [url]
```

3. Create a Branch: Developers create a new branch for their feature or bug fix. This is usually done using the git checkout -b command.
Bash

```
git checkout -b feature-branch
```

4. Make Changes and Commit: Developers make changes to the code and add and commit those changes using the standard Git commands.

```
git add .
git commit -m [descripttive message]
```

5. Push the Branch to GitHub: Developers push the newly created branch and their changes to their GitHub fork.

```
git push [alias] [branch-name]
```

6. Create a Pull Request (PR): Developers go to their GitHub repository and create a pull request from their feature branch to the original repository's master branch.

7. Code Review: Team members or collaborators review the changes in the pull request, provide feedback, and discuss the proposed changes.

8. Address Feedback: Developers make additional changes to their branch based on the feedback received during the code review process.

9. Merge the Pull Request: Once the changes have been reviewed and approved, the pull request is merged into the master branch of the original repository.

10. Delete the Branch: After the merge is complete, the feature branch is typically deleted both locally and on the GitHub repository.

Q. Explain the git diff command, and why it is used for?
Ans:
The git diff command in Git is used to display the differences between two states of a repository. It provides a way to compare changes in various contexts, such as between the working directory and the staging area, between the working directory and the last commit, or between two different commits or branches.

Here are some common use cases for the git diff command:

1. **Compare Working Directory with Staging Area:** To see the changes that have been made but not yet staged, you can use:

```
git diff
```

2. **Compare Working Directory with Last Commit:** To see the differences between the working directory and the last commit, you can use:

```
git diff HEAD
```

3. **Compare Staging Area with Last Commit:** To see the changes that are staged but not yet committed, you can use:

```
git diff --staged
```

4. **Compare Two Commits:** To compare changes between two specific commits, you can provide the commit hashes:

```
git diff commit-hash-1 commit-hash-2
```

5. **Compare a File Across Commits:** To see changes in a specific file between two commits, you can specify the file path:

```
git diff commit-hash-1 commit-hah-2 -- file-path
```

6. **Viewing Changes in a Specific Branch:** You can compare the changes between the current branch and another branch:

```
git diff branch-name
```

The output of the **git diff** command provides a line-by-line representation of the changes, indicating additions with a + symbol and deletions with a - symbol. It helps developers understand what has been modified, added, or deleted in the codebase.

By using **git diff**, developers can review changes before committing them, identify issues, and ensure that the modifications align with their intentions. It's a powerful tool for tracking changes at various stages of the development process and is crucial for maintaining code quality and consistency in collaborative projects.

Q. Explain the CI/CD pipeline integration with Git
Ans -
The integration of a CI/CD pipeline with Git involves leveraging Git's version control capabilities to automate the software development lifecycle, from code changes to deployment. Here's an explanation based on the provided sources:

- CI/CD is a methodology used to automate the software development lifecycle, ensuring frequent and reliable updates are delivered quickly and efficiently.
- Continuous Integration (CI) involves triggering automated builds for every code change to ensure functionality and catch bugs early on.
- Continuous Delivery (CD) automates the deployment of successfully tested changes to production environments, reducing the gap between development and release.
- GitHub Actions allows you to respond to any webhook on GitHub, enabling automation based on various triggers like pull requests, issues, and comments.
- Community-powered, reusable workflows in GitHub Marketplace provide access to over 11,000 actions, promoting consistency and reducing duplication.
- GitHub Actions is platform, language, and cloud agnostic, allowing flexibility in technology choices.