Name :- Vijay Gurung

ID Number : MST03 – 0070

Date : 05-July-24

**Topic : * Git_&_GitHub_Test_Questions *

Q1. What is Git and why is it used?

Answer -

Git is a version control system that is used to track changes to our files. It is a free and open-source software that is available for Windows, macOS and Linux. Git was created by Linus Torvalds in 2005.

Before **Git** became mainstream, version control systems were used by developers to manage their code. They were called SCCS (Source Code Control System). SCCS was a proprietary software that was used to manage the history of code. It was expensive and not very user-friendly. Git was created to replace SCCS and to make version control more accessible and user-friendly. Some commong version control systems are Subversion (SVN), CVS and Perforce.

Git Used are as follows :

1.Git maintains a history of changes, making it easy to revert to previous versions if something goes wrong.

2. Multiple developers can work on the same project simultaneously, with changes merged systematically.

3.Developers can create branches to work on new features or fixes independently. These branches can later be merged back into the main codebase.

4.Every developer has a local copy of the entire project history, allowing for offline work and contributing to a more robust system.

5.Git tracks who made changes, what was changed and why making it easier to manage and understand the evolution of the project.Because every developer has a full copy of the repository, it acts as a backup of the project.

Q2.Explain the difference between Git pull and Git fetch ?

Answer :

The main difference between **Git Pull and Git Fetch** is that **git pull** automatically merges the fetched changes into our current local branch.

while **git fetch** only downloads the new data from the remote repository without integrating it into our working files.

Below is an illustration of the differences :

For example we have a remote repository named **“Origin”** and a local branch named **“Master”**. The remote repository has some new commits that we don't have in our local repository.

1.Git_Fetch :

(A) Run git fetch origin to download the new commits from the remote repository without merging them into our local branch.

(B) After running git fetch,we can use **“ git log --oneline --graph --decorate --all ”** to see the commit history. We will notice that our local master branch hasn't moved, but we can see the new commits in the remote **“origin/master”** branch.

(C) If we want to integrate the fetched changes into our local branch, we can run **“git merge origin/master”** to merge the **“origin/master”** branch into our local master branch.

2.Giit_Pull :

(A) Run git pull to download the new commits from the remote repository and automatically merge them into our local master branch.

(B) After running git pull, our local **“master”** branch will be updated with the new commits from the remote repository.

Git Fetch downloads the new data from the remote repository, but doesn't integrate it into our working files.

while **Git Pull** downloads the new data and automatically merges it into your current local branch.

Using git fetch allows us to review the changes before merging them into our local branch, while **Git Pull** is more convenient when we want to quickly update into our local branch with the latest changes from the remote repository.

Q3.How do you revert a commit in Git?

Answer :

To revert a commit in Git, We can use the “**git revert”** command. This command creates a new commit that undoes the changes made in the specific commit.

Below are the details explained :

For example let's say we open our terminal or command prompt and navigate to our G**it repository.**

Run the following command to revert a specific commit :

[git revert <commit_hash>]

Replace **“ <commit_hash> ”** with the hash of the commit we want to revert. We can find the “**commit hash using git log”**.

Git will open our default text editor. Add a commit message describing the revert and save the file.

Git will create a new commit that undoes the changes made in the specific commit.

If there are any conflicts during the revert process, Git will pause and ask us to resolve the conflicts manually. After resolving the conflicts, stage the resolved files using git add and continue the revert process using

[git revert --continue]

For Example :

Let's say we have the following commit history :

*** 5a9da2f (HEAD -> master) Add feature X**

*** 8b6acf1 Fix bug in feature Y**

*** 2c14b8e Implement feature Y**

*** 1a3c5f2 Initial commit**

And we want to revert the commit **"Add feature X" (hash: 5a9da2f)**. Run the following command:

[git revert 5a9da2f]

Git will create a new commit that undoes the changes made in the **"Add feature X"** commit:

*** 2a1b3c4 (HEAD -> master) Revert "Add feature X"**

*** 5a9da2f Add feature X**

*** 8b6acf1 Fix bug in feature Y**

*** 2c14b8e Implement feature Y**

*** 1a3c5f2 Initial commit**

The new commit **"Revert 'Add feature X'"** will have the opposite changes of the reverted commit, effectively undoing its changes.

Note :

Git revert creates a new commit to undo the changes, rather than removing the commit from the history. This is a safer approach compared to git reset, which can potentially cause issues if we have already pushed the commit to a remote repository.

4.Describe the Git staging area ?

Answer :

The Git staging area :

It is also known as the **“index”**, a crucial concept in **Git.** It acts as an intermediate area between our working directory and the repository.

The staging area allows us to selectively add changes from our working directory to the next commit. This gives us more control over what goes into each commit. Instead of committing all the changes we have made, we can choose to include only specific modifications in the next commit.

Lets understand how the Git staging area works:

You make changes to files in your working directory.

You use the git add command to add the changes to the staging area.

When you're ready, you run git commit to create a new commit based on the changes in the staging area.

For Example:

Let's say we have a directory with three files: **“file1.txt, file2.txt, and file3.txt”**. We make changes to all the 3 files in our working directory.

[Make changes to file1.txt, file2.txt, and file3.txt]

Now, Wou can selectively add the changes to the staging area using **“git add”:**

Add changes to file1.txt and file3.txt to the staging area

[git add file1.txt file3.txt]

The changes to file2.txt are still in the working directory but not in the staging area

The changes to file2.txt are still in the working directory but not in the staging area

After adding the desired changes to the staging area, We can create a **new commit:**

Create a new commit with the changes from the staging area

[git commit -m "Update file1.txt and file3.txt"]

The new commit will only include the changes to **“file1.txt and file3.txt”** that were added to the staging area. The changes to **file2.txt** remain in the working directory and will not be part of this commit.

The staging area allows us to organize our changes, test them and create meaningful commits before pushing them to the remote repository. It's a powerful feature that gives us more control over the commit history and helps maintain a clean and organized **Git repository**.

5.What is a merge conflict and how can it be resolved?

Answer :

A **merge conflict** occurs when **Git** is unable to automatically merge changes from two different branches. This typically happens when two branches have made changes to the same line(s) of code and Git doesn't know which version to keep.

Below are steps how to resolve a merge conflict:

(A) When a merge conflict occurs, Git will mark the conflicting sections in the affected files. We can identify these files using git status.

(B) Locate the conflicting files and open them in a text editor. We will see the conflicting changes marked with special conflict markers:

<<<<<<< HEAD

 # Your changes

=======

 # Changes from the other branch

>>>>>>> other-branch

(C) Manually edit the conflicting sections to keep the desired changes. Remove the conflict markers and keep only the code we want to retain.

(D) After resolving the conflicts, stage the changes using **" git add <file>"** for each resolved file.

(E) Once all conflicts are resolved and staged, run git commit to complete the merge. Git will create a new merge commit with the resolved conflicts.

(F) If we were merging a remote branch, push the resolved conflicts to the remote repository using **"git push".**

Below is an example of how we can resolving a merge conflict:

Merge a remote branch

[git merge origin/feature-branch]

Identify conflicting files

[git status]

Unmerged: file1.txt

Open file1.txt and resolve conflicts

<<<<<<< HEAD

Your local changes

 # =======

Changes from remote branch

>>>>>>> origin/feature-branch

Stage resolved conflicts

[git add file1.txt]

Complete the merge

[git commit -m "Resolve merge conflict in file1.txt"]

Push the resolved conflicts

[git push]

So, by following these steps, We can successfully resolve merge conflicts and complete the merge process.

6.How does Git branching contribute to collaboration?

Answer :

“Git branching” is a fundamental feature that enables effective collaboration in software development projects. By allowing developers to work on separate branches, branching facilitates parallel development, experimentation and the ability to incorporate changes without disrupting the main codebase.

Below are the details explained that how Git branching contributes to collaboration:

(A) Developers can work on different features or bug fixes simultaneously by creating separate branches. This allows the team to make progress on multiple tasks in parallel without interfering with each other's work.

(B) Branches provide an isolated environment for developers to experiment with new ideas, test features, or fix bugs without affecting the main codebase. If an experiment fails, the branch can be discarded without impacting the production-ready code.

(C) When a developer completes their work on a branch, they can create a pull request (or merge request) to merge their changes into the main branch. This allows other team members to review the code, provide feedback and ensure code quality before integrating the changes.

(D) Branches enable incremental deployments, where features or bug fixes can be deployed to production individually. This allows for faster delivery of value to customers and reduces the risk of large, monolithic deployments.

(E) Multiple developers can collaborate on a single feature by working on the same branch. They can coordinate their efforts, resolve conflicts, and ensure consistency in the implementation.

(F) If a critical bug is discovered in production, a hotfix branch can be created from the production branch. Developers can work on fixing the bug in isolation while the main development continues on other branches.

For Example :

Let's say a team is working on a web application. They have a main branch called master that represents the production-ready code. Each developer creates a new branch for their assigned tasks:

feature/login: Implement user login functionality

feature/registration: Implement user registration functionality

bugfix/404-error: Fix a 404 error on the contact page

The developers work on their respective branches, committing changes and pushing their branches to the remote repository. When a feature is complete, the developer creates a pull request to merge their branch into master. Other team members review the code and provide feedback. Once approved, the changes are merged and deployed to production.

If a critical bug is discovered in production, a hotfix branch is created from master:

[hotfix/critical-bug: Fix a critical bug in production]

The developer fixes the bug on the **"hotfix/critical-bug"** branch, creates a pull request and merges the changes into master. The master branch is then deployed to production, resolving the critical issue.

By leveraging **Git branching**, the team can collaborate effectively, work on multiple tasks simultaneously, experiment with new ideas and ensure code quality through code reviews, all while maintaining a stable production environment.

7. What is the purpose of Git rebase?

Answer :

The **“purpose of git rebase”** is to integrate changes from one branch into another branch. It allows us to reapply commits from one branch on top of another branch, effectively re-writing the commit history.

Below is the how git rebase works:

(A) We have two branches: **“feature and main”.**

(B) The main branch has progressed with new commits while we were working on the **“feature branch”.**

(C) To integrate the changes from main into feature, we run **“git rebase main”** from the **“feature branch”.**

(D) **Git will:**

- Find the common ancestor commit between feature and main.
- Temporarily store the changes introduced by each commit on feature since the common ancestor.
- Reset the feature branch to the same commit as main.
- Reapply each commit from the stored changes on top of main.

For Example :

Let's say we have the following commit history:

*** 7a1d1f5 (main) Add new feature**

*** 3c2d6f1 Fix bug**

*** 9a4c2b3 Refactor code**

*** 5e3c1a4 (feature) Initial commit**

we create a new branch feature and make some changes:

*** 7a1d1f5 (main) Add new feature**

*** 3c2d6f1 Fix bug**

*** 9a4c2b3 Refactor code**

*** 5e3c1a4 (feature, origin/feature) Initial commit**

*** 2b3c4d5 Add new functionality**

*** 1c2d3e4 Fix bug in feature**

Meanwhile, the main branch has progressed with new commits:

*** 7a1d1f5 (main) Add new feature**

*** 3c2d6f1 Fix bug**

*** 9a4c2b3 Refactor code**

*** 1d2e3f4 Improve performance**

*** 5a6b7c8 Fix security issue**

To integrate the changes from main into feature, you run git rewrite from the feature branch:

$ git checkout feature

$ git rebase main

Git will reapply the commits from feature on top of main, resulting in:

(feature)

2b3c4d5 Add new functionality

1c2d3e4 Fix bug in feature

*** 7a1d1f5 (main) 1d2e3f4 Improve performance**

*** 3c2d6f1 5a6b7c8 Fix security issue**

*** 9a4c2b3 Add new feature**

*** 1d2e3f4 Fix bug**

*** 5a6b7c8 Refactor code**

*** 5e3c1a4 Initial commit**

The commits from feature are now on top of main, effectively integrating the changes.

Conclusion :

By using git rebase, We can keep your feature branches up-to-date with the main branch and maintain a linear commit history. However, it's important to use git rebase carefully, especially on public branches, as it can cause issues if we rebase commits that have already been pushed to a remote repository.

8. Explain the difference between Git clone and Git fork?

Answer :

The main difference between git clone and git fork is that git clone creates a local copy of an existing repository on your machine, while git fork creates a copy of the repository on a different server, usually on a remote hosting service like GitHub.

1.Git clone :

- git clone creates a local copy of an existing repository on your machine.
- The cloned repository is linked to the original repository, allowing you to fetch updates and push changes.
- When you clone a repository, you become a collaborator on the project.

For Example: [git clone https://github.com/user/repository.git]

2.Git fork :

- Git fork creates a copy of the repository on a remote hosting service like GitHub.
- The forked repository is a separate copy of the original repository, not directly linked to it.
- When we fork a repository, we create your own version of the project under our account.
- We can then clone our forked repository to our local machine and make changes.

For Example: [On GitHub, We can fork a repository by clicking the **"Fork"** button on the repository page]

Below is a short example to illustrate the difference:

Let's say there's a repository called **"project"** on GitHub owned by **"user1"**. We want to contribute to this project.

Using git clone:

We are added as a collaborator to "user1/project".

We run git clone https://github.com/user1/project.git on your local machine.

We make changes to the cloned repository and push them back to "user1/project".

Using git fork:

We fork "user1/project" by clicking the "Fork" button on GitHub.

GitHub creates a copy of "user1/project" under our account as "our-username/project".

We clone our forked repository to your local machine using git clone

[https://github.com/your-username/project.git]

You make changes to the cloned repository and push them to "our-username/project".

When we want to contribute to the original project, you create a pull request from your forked repository to "user1/project".

Conclusion :

Git clone is used to create a local copy of an existing repository, while git fork is used to create a copy of a repository on a remote hosting service, allowing us to work on our own version of the project and contribute back to the original repository through pull requests.

9.How do you delete a branch in Git?

Answer :

To delete a branch in Git, you can use the git branch -d command followed by the name of the branch you want to delete. Here's how it works:

- We can list all the branches in your repository using git branch. This will show us all the local branches.
- To delete a local branch, run git branch -d <branch-name>. Replace <branch-name> with the name of the branch we want to delete.

For example : **[git branch -d feature/login]**

- Delete a remote branch: To delete a remote branch, you can use git push <remote> --delete <branch-name>. Replace <remote> with the name of your remote repository (e.g., origin) and <branch-name> with the name of the branch you want to delete.

For example: **[git push origin --delete feature/login]**

This will delete the feature/login branch from the remote repository.

<u>For Example:</u>

Let's say we have the following branches in your repository:

main

*** feature/login**

feature/registration

bugfix/404-error

The * indicates the currently checked-out branch (feature/login).

To delete the feature/login branch, we can run:

[git branch -d feature/login]

Git will prompt you to confirm the deletion. If the branch has already been merged into another branch, Git will delete the branch. If the branch has unmerged changes, Git will prevent us from deleting it by default.

If we want to force the deletion of an unmerged branch, you can use git branch -D <branch-name> (note the uppercase D).

After deleting the branch, our branch list will look like this:

main

feature/registration

bugfix/404-error

Note :

Deleting a branch permanently removes it from our local repository. If we have already pushed the branch to a remote repository, we will need to delete it from the remote as well using

[git push <remote> --delete <branch-name>]

10. What is a Git hook, and how can it be used?

Answer :

A **Git hook** is a script that Git executes before or after an event, such as a commit, push or receive. Git hooks are a powerful feature that allows us to customize the behavior of our Git repository and automate certain tasks.

Git hooks are stored in the .git/hooks directory of your repository. When a specific event occurs, Git looks for a corresponding hook script in this directory and executes it if it exists.

Here are some examples of how Git hooks can be used:

- ★ **Pre-commit hook:** This hook runs before a commit is created. We can use it to check for common coding standards, run tests, or prevent committing certain files.

Example:-

- ★ A pre-commit hook that runs eslint to check for JavaScript code style violations:

```
#!/bin/sh

eslint .

if [ $? -ne 0 ]; then

  echo "ESLint failed. Please fix the issues before committing."

  exit 1

fi
```

- ★ **Post-commit hook:** This hook runs after a commit is created. You can use it to notify team members about the new commit or perform other post-commit tasks.

Example: A post-commit hook that sends a notification to a Slack channel:

```
#!/bin/sh

slack_webhook="https://hooks.slack.com/services/YOUR_WEBHOOK_URL"

commit_message=$(git log -1 --pretty=%B)

curl -X POST -H 'Content-type: application/json' --data "{\"text\":\"New commit: $commit_message\"}" $slack_webhook
```

- ★ **Pre-push hook:** This hook runs before pushing changes to a remote repository. You can use it to run tests, check for sensitive information in the commits, or enforce certain rules before pushing.

Example: A pre-push hook that runs pytest to ensure all tests pass before pushing:

```
#!/bin/sh

pytest

if [ $? -ne 0 ]; then

  echo "Tests failed. Please fix the issues before pushing."

  exit 1

fi
```

- ★ **Prepare-commit-msg hook:** This hook runs before the commit message editor is displayed. You can use it to pre-populate the commit message with information like the ticket number or a template.

Example: A prepare-commit-msg hook that prepends the branch name to the commit message:

```
#!/bin/sh

branch_name=$(git rev-parse --abbrev-ref HEAD)

sed -i.bak -e "1i\

# $branch_name

" $1
```

Conclusion :

These are just a few examples of how Git hooks can be used. We can create custom hooks for our specific needs and automate various tasks in our Git workflow.

Hook scripts must be executable (e.g., chmod +x .git/hooks/pre-commit) for Git to run them.