**Assignments 3 Swapnil Gaikwad**

1. What is the difference between a hard reset, a soft reset, and a mixed reset in Git?

Answer –

**Hard Reset**

* In hard reset HEAD simply points to an older commit and we can see the data as present at the time of that older commit
* It will delete data from working directory.
* Syntax:

git reset --hard commit\_id

**Soft Reset**

* Soft reset will move the head to an older commit.
* Files will be present in the staging area.
* Syntax:

git reset --soft commitid

**Mixed Reset**

* Mixed reset moves the head to an older commit.
* The files will be present in the untracked/modified section

git reset --mixed commitid

1. How would you bring files from the staging area back to the working directory without losing changes?

Answer -

To bring files from the staging area back to the working directory without losing any changes, you can use the git reset command. This will unstage the files but keep your modifications in the working directory intact.

1. What is a Git tag?

Answer - Git tags are aliases for commit ids. Tags are always created against specific commit ids

1. How do you create a tag for a specific commit?

Answer -

* git tag --a v1.3 <commit id> -m "any message"

1. Explain how to list all available tags in a repository.

Answer -

lists all available tags

git tag

1. What is the purpose of creating branches in Git?

Answer – In Git, branches are used to isolate work, allowing you to make changes in a separate line of development without affecting the main or other existing codebases.

1. Parallel Development

Purpose: Branches allow multiple developers (or teams) to work on different features, bug fixes, or tasks independently without affecting the main codebase (typically the main or master branch).

2. Isolating Features and Changes

Purpose: By creating a branch for each feature or task, you can isolate changes related to that specific feature from other parts of the code.

3. Easy Collaboration

Purpose: In teams, branches allow developers to work independently on different parts of a project, then merge their changes back into the main branch when they are ready.

4. Version Control for Features or Tasks

Purpose: Branches provide a convenient way to develop and manage new features or tasks separately from the main production code.

5. Easier Bug Fixing

Purpose: If an issue arises, you can create a branch specifically to address the bug, work on a fix, and then merge the fix back into the main branch.

6. Clean Merging of Changes

Purpose: Once development is complete on a feature or task branch, you can merge it back into the main branch (e.g., main or develop).

7. Experimentation and Prototyping

Purpose: Branches allow developers to experiment with new features, technologies, or ideas without affecting the main codebase.

8. Maintaining Stable Releases

Purpose: Branches like master or main often represent the stable production version, while other branches (e.g., develop) may represent ongoing development work.

1. How do you create a new branch and switch to it in a single command?

Answer - use cmd-

git checkout -b <new-branch>

1. Explain the steps to delete a branch from local repo.

Answer -

1. Check the Current Branch

git branch

2. Switch to a Different Branch

git checkout main

3. Delete the Local Branch

git branch -d <branch-name>

1. What is the purpose of the git merge command?

Answer -

The purpose of the git merge command is to combine the changes from one branch into another. It is primarily used to integrate different branches of development, such as merging a feature branch back into the main or master branch once development is complete.

1. Explain the difference between a standard git merge and a git merge --squash.

Answer -

**Key Differences:**

| **Aspect** | **git merge** | **git merge --squash** |
| --- | --- | --- |
| **Commit History** | Keeps all individual commits from the source branch. | Squashes all commits from the source branch into a single commit. |
| **Merge Commit** | Creates a merge commit with two parents. | Does not create a merge commit. |
| **Resulting Commit** | Adds multiple commits to the target branch. | Adds one commit (squashed) to the target branch. |

1. After performing a git merge --squash, what steps must you take to complete the merge?

Answer -

Stage the Changes (if necessary):

git add .

Commit the Squashed Changes:

git commit -m "Merge feature-branch as a single commit"

Verify the Merge:

git status

git log

1. Explain about git merge with rebase.

Answer -

git merge with rebase refers to a workflow where instead of merging a branch into another (which creates a merge commit), you reapply the commits from one branch onto another as if they happened after the latest commit of the target branch.

- Rebase takes all the changes from a feature branch and \*\*applies them\*\* onto the base branch (e.g., `main`).

- This rewrites the commit history by placing the commits from the feature branch on top of the current branch (usually `main`), making the history linear.

How `git merge --rebase` Works:

1. When you rebase, Git temporarily stashes your changes.

2. It updates your current branch with the latest commits from the target branch (e.g., `main`).

3. Finally, Git re-applies your commits from the feature branch, as if you had made them on top of the latest changes from the target branch.

Key Points:

-No Merge Commit: Unlike a regular merge, rebase does not create a merge commit.

- Linear History: It helps keep the project history linear, which can make it easier to read and understand.

- Potential Conflicts: Conflicts may occur when reapplying your changes, and you'll need to resolve them.

Example:

Suppose you have a `main` branch and a `feature-branch`. If you run:

git checkout feature-branch

git rebase main

This will:

1. Update `feature-branch` with the latest changes from `main`.

2. Apply the commits from `feature-branch` on top of `main`, making the history appear as if the feature branch started from the latest commit on `main`.

When to Use Rebase:

- Use rebase when you want to keep a clean, linear history and avoid merge commits.

- It’s often used in feature branches before merging them back into the main branch.