**Git Assessment-2**

1. **How do you rename the current branch you are on to a new-branch-name?**

git branch -M new-branch-name

git branch: This tells Git that we're working with branches.

-M: This flag is used to move or rename a branch.

new-branch-name: This is the new name you want to give to the current branch.

1. **What is the command to stash your changes and include untracked files?**

git stash -u

git stash: This command saves changes to a stack, so you can come back to them later.

-u: This flag includes untracked files in the stash.

1. **How would you merge changes from the branch feature-new into your current branch?**

git merge feature-new

git merge: This command merges changes from one branch into another.

feature-new: This is the name of the branch you want to merge into your current branch

1. **What command do you use to view the differences between your working directory and the last commit?**

git diff

1. **Explain how to resolve a merge conflict that occurs during a git merge. What steps would you take?**
2. Identify the Conflicts: Git will mark conflicting files.
3. Edit the Files: Manually resolve the conflicts in a text editor.
4. Stage the Resolved Files: Use “git add <filename>” to stage each resolved file.
5. Commit the Merge: Use “git commit” to complete the merge.
6. **Explain the purpose and use case of git rebase with an example. How does it differ from git merge?**

Git Rebase: Re-applies commits on top of another branch, creating a linear history.

Use Case: When you want a clean, linear history.

Git Merge: Merges two branches into one, creating a merge commit.

Use Case: When you want to preserve the history of both branches.

1. **What does the command git cherry-pick <commit hash> do? Provide a scenario where you might use it.**
2. **Describe the differences between git reset --soft, git reset --mixed, and git reset --hard. When would you use each?**
3. git reset --soft: Resets the HEAD pointer but keeps the changes in the staging area.
4. git reset --mixed: Resets the HEAD pointer and removes changes from the staging area, but keeps the changes in the working directory.
5. git reset --hard: Resets the HEAD pointer, removes changes from the staging area, and discards changes in the working directory.
6. **Describe how you would revert a commit that has already been pushed to a shared repository. What command would you use, and what should you consider before doing this?**
7. **What is the difference between git pull --rebase and git pull? When would you prefer one over the other?**

git pull and git pull --rebase are both commands used to update your local branch with changes from a remote repository. However, they differ in how they integrate these changes.

git pull

This command merges the remote changes into your local branch, creating a new merge commit.

Over time, this can lead to a more complex and less linear commit history, especially with frequent merges.

git pull --rebase

This command replays your local commits on top of the remote branch, creating a cleaner, more linear history. By rebasing, you avoid creating unnecessary merge commits.

**When to use which**

**git pull**

1. If you're working in a team and need to merge changes frequently, using `git pull` can be a straightforward way to integrate changes.
2. If you're unsure about resolving rebase conflicts or if you're working with a large team, `git pull` might be a safer option.

**git pull –rebase**

1. If you prefer a clean and linear commit history, rebasing can help you achieve that.
2. Before creating a pull request, rebasing your local branch onto the remote branch can make the review process smoother.
3. For personal projects where you're the sole developer, rebasing can simplify the commit history.

**Important Considerations**

1. Once a commit has been pushed to a shared repository, rebasing it can cause conflicts and confusion for other developers.
2. Both `git pull` and `git pull --rebase` can lead to conflicts. You'll need to resolve these conflicts manually before proceeding.
3. If you rebase a branch that has already been pushed to a remote repository, you'll need to force push your changes, which can overwrite the history for other developers. This should be done with caution and only when necessary.
4. **How can you view the changes made by a specific commit? What command do you use?**

git show <commit\_hash>

Replace <commit\_hash> with the actual hash of the commit you want to inspect.

1. The message associated with the commit.
2. The author of the commit and the date it was committed.
3. A detailed diff showing the specific changes made to each file in the commit.
4. **What is branch protection in Git, and why is it important for a collaborative development environment?**

Branch Protection is a feature in Git that helps to safeguard critical branches, such as the main or development branch, from accidental or malicious changes. It imposes specific rules and restrictions on these branches to ensure code quality and stability.

Why it is important

1. It prevents accidental pushes to protected branches, such as deleting important files or committing broken code.
2. It can require code reviews before changes are merged into protected branches, ensuring that multiple people have reviewed the code and agreed to its quality.
3. It can enforce specific checks, such as running tests or linting, before merging changes. This helps maintain code quality and consistency.
4. It can prevent unauthorized users from pushing malicious code to protected branches.
5. **Explain how to set up branch protection rules in GitHub.**
6. **What happens if a user tries to push changes directly to a protected branch?**
7. **What is the purpose of a .gitignore file, and how do you create one? Provide an example of what you might include in it.**