ALL ABOUT GIT

Git is a powerful version control system that allows you to manage and track changes to your codebase. Here are some commonly used Git commands:

1. `git init`: Initializes a new Git repository in the current directory.

2. `git clone <repository>`: Creates a local copy of a remote repository.

3. `git add <file>`: Adds a file or directory to the staging area.

4. `git commit -m <message>`: Commits the changes in the staging area with a descriptive message.

5. `git status`: Shows the current state of the repository, including modified files and untracked files.

6. `git diff`: Displays the differences between the working directory and the staging area.

7. `git diff --staged`: Shows the differences between the staging area and the last commit.

8. `git push`: Pushes the local commits to a remote repository.

9. `git pull`: Fetches and merges changes from a remote repository into the current branch.

10. `git branch`: Lists all branches in the repository.

11. `git branch <branch\_name>`: Creates a new branch.

12. `git checkout <branch\_name>`: Switches to the specified branch.

13. `git merge <branch\_name>`: Merges the specified branch into the current branch.

14. `git remote add <name> <url>`: Adds a new remote repository.

15. `git log`: Displays the commit history.

16. `git reset <file>`: Removes a file from the staging area.

17. `git reset --hard <commit>`: Resets the repository to a specific commit, discarding any subsequent changes.

18. `git stash`: Stashes changes in a temporary area to allow you to switch branches without committing.

19. `git tag <tag\_name>`: Creates a new tag at the current commit.

20. `git show <commit>`: Displays the details of a specific commit.

To undo a commit in Git, you have a few options depending on your specific situation. Here are some common scenarios and the corresponding commands to undo a commit:

* Undo the last commit and keep the changes as unstaged

**git reset HEAD~**

This command moves the HEAD pointer of your current branch to the previous commit, effectively undoing the last commit. The changes from the undone commit will still be present in your working directory, allowing you to modify and recommit them as needed.

2. Undo the last commit and discard the changes:

git reset --hard HEAD~

This command not only moves the HEAD pointer to the previous commit but also discards all the changes introduced by the undone commit. Use this command with caution as it permanently deletes the changes.

3. Undo a commit that has been pushed to a remote repository:

If you have already pushed a commit and want to undo it, you should avoid modifying the commit history directly as it can cause issues for other collaborators. Instead, you can use the `git revert` command to create a new commit that undoes the changes introduced by the previous commit. For example:

**git revert <commit\_sha>**

Replace `**<commit\_sha>`** with the commit hash of the commit you want to revert. Git will create a new commit that undoes the changes, and you can push this new commit to the remote repository.

When Git encounters conflicting changes during a merge, it generates a merge conflict. Resolving merge conflicts involves manually editing the conflicting files to reconcile the differences. Here's a step-by-step guide to solving merge conflicts in Git:

1. Identify the conflicts: When you attempt to merge branches and conflicts arise, Git will notify you of the conflicting files. You can use the command `git status` to see which files have conflicts.

2. Open the conflicting file(s): Use a text editor to open the conflicting file(s). Git inserts markers to indicate the conflicting sections. The markers look like this:

**```**

**<<<<<<< HEAD**

**// Your changes**

**=======**

**// Incoming changes**

**>>>>>>> branch\_name**

**```**

The section between `<<<<<<< HEAD` and `=======` represents the changes in your current branch, while the section between `=======` and `>>>>>>> branch\_name` represents the changes in the branch you are merging.

3. Resolve the conflicts: Review the conflicting sections and decide how to reconcile the differences. You can choose to keep your changes, discard them in favor of the incoming changes, or manually edit the sections to create a new version that combines both sets of changes.

4. Remove conflict markers: After resolving the conflicts, remove the conflict markers (`<<<<<<< HEAD`, `=======`, `>>>>>>> branch\_name`) from the file. Ensure that the final version represents the desired changes.

5. Add the resolved files: Once you have resolved the conflicts in a file, stage the modified file by using the command `git add <file>`.

6. Commit the merge: After adding all resolved files, create a new commit to complete the merge by using the command `git commit`. Git will automatically generate a commit message indicating that a merge conflict was resolved.

7. Complete the merge: If you are performing a regular merge, use `git merge --continue` after committing the changes to finalize the merge. If you are using the `git pull` command to fetch and merge changes, Git will handle the final steps automatically.

* To show which branches are to be merged and which files are already merged, you can use the following Git commands:

1. Show branches to be merged:

**```**

**git branch --no-merged**

**```**

This command lists all branches that have commits not yet merged into the current branch. It displays the branches that are candidates for merging into the current branch.

2. Show branches already merged:

**```**

**git branch --merged**

**```**

This command lists all branches that have been fully merged into the current branch. It displays the branches whose changes are already incorporated into the current branch.

3. Show files already merged:

**```**

**git diff --name-only --diff-filter=U**

**```**

To unstage and untrack a file in Git, you can use the following commands:

1. Unstage a file:

**```**

**git reset HEAD <file>**

**```**

This command removes the file from the staging area, but preserves its modifications in the working directory. Replace `<file>` with the path to the file you want to unstage.

2. Untrack a file:

If you want to completely remove a file from Git's tracking, including removing it from both the staging area and the repository, you can use the following steps:

a. First, remove the file from the staging area using the command mentioned above to unstage the file:

**```**

**git reset HEAD <file>**

**```**

b. Then, remove the file from the repository and working directory using the `git rm` command:

**```**

**git rm --cached <file>**

**```**

This command removes the file from the repository while keeping it in the working directory. The `--cached` option ensures that the file is only removed from the repository and not deleted from the disk.

c. Finally, you should commit the changes to complete the removal of the file from tracking:

**```**

**git commit -m "Remove <file>"**

**```**

Replace `<file>` with the path to the file you want to untrack.

Git rebase is a powerful Git command that allows you to apply the changes from one branch onto another. It is often used to incorporate changes from one branch into another branch while maintaining a linear commit history.

The basic syntax of the `git rebase` command is as follows:

```

git rebase <base\_branch>

```

Here's how you can perform a rebase in Git:

1. Ensure you are on the branch where you want to apply the changes:

**```**

**git checkout <target\_branch>**

**```**

2. Run the rebase command, specifying the branch from which you want to apply the changes:

**```**

**git rebase <source\_branch>**

**```**

Replace `<source\_branch>` with the branch containing the changes you want to apply.

The rebase operation takes the commits from the `<source\_branch>` that are not present in `<target\_branch>`, detaches them from their original branch, and applies them one by one on top of `<target\_branch>`. This process incorporates the changes from `<source\_branch>` onto `<target\_branch>`.

3. Resolve any conflicts:

If Git encounters conflicts during the rebase, it will pause the process and notify you. You need to resolve the conflicts manually by editing the conflicting files, marking the conflicts as resolved, and staging the changes. Use `git status` to see the conflicting files and `git diff` to inspect the conflict markers.

After resolving conflicts in each file, use the following command to continue the rebase:

**```**

**git rebase --continue**

**```**

If you want to abort the rebase at any point, you can use:

**```**

**git rebase --abort**

**```**

This command will revert the branch to its original state before the rebase was started.

4. Push the updated branch:

Once the rebase is complete, you may need to force-push the updated branch to the remote repository if you have already pushed the original version of the branch. Use the following command to force-push:

**```**

**git push --force**

**```**

Note that force-pushing rewrites the branch's history, so be cautious when using it in a collaborative environment.

* A `.gitignore` file is used to specify files and directories that should be ignored by Git when tracking changes in a repository. It allows you to exclude certain files, such as build artifacts, temporary files, or sensitive data, from being committed. You can create a `.gitignore` file in the root directory of your Git repository and list the files and patterns you want to ignore. Git will then disregard those files when staging and committing changes.
* To skip the staging area and directly commit changes in Git, you can use the `-a` or `--all` option with the `git commit` command. This option allows you to automatically stage and commit all tracked files that have been modified or deleted.
* To move and rename files in Git, you can use the `git mv` command. It is a convenience command that combines the steps of moving a file and renaming it in one operation.
* To view and change commits in Git, you can use various commands and techniques. Some commonly used commands include `git log` to view the commit history, `git show` to display the details of a specific commit, `git commit --amend` to modify the most recent commit, and `git rebase -i` to interactively modify older commits.

To push a local branch to a remote GitHub repository, you can use the `git push` command. Here's how you can do it:

1. Ensure that you have a local Git repository set up and have the necessary permissions to push to the remote GitHub repository.

2. Open a terminal or command prompt and navigate to your local repository's root directory.

3. Verify the remote repositories associated with your local repository by running the following command:

**```shell**

**git remote -v**

**```**

This command will display the names and URLs of the remote repositories.

4. If the remote GitHub repository is not already added as a remote, you can add it using the `git remote add` command:

**```shell**

**git remote add origin <remote\_repository\_url>**

**```**

Replace `<remote\_repository\_url>` with the URL of the remote GitHub repository. The name "origin" is commonly used to refer to the main remote repository, but you can choose a different name if desired.

5. Ensure that your local branch is up to date with the latest changes from the remote repository by running the following command:

**```shell**

**git pull origin <remote\_branch\_name>**

**```**

Replace `<remote\_branch\_name>` with the name of the branch you want to push. This step helps avoid conflicts and ensures that you have the latest changes before pushing your branch.

6. Once you have the latest changes, you can push your local branch to the remote GitHub repository using the `git push` command:

**```shell**

**git push origin <local\_branch\_name>:<remote\_branch\_name>**

**```**

Replace `<local\_branch\_name>` with the name of your local branch, and `<remote\_branch\_name>` with the desired name for the branch on the remote GitHub repository. If the branch doesn't exist on the remote repository, it will be created.

Alternatively, if you want to push the branch with the same name on both local and remote repositories, you can run:

**```shell**

**git push origin <branch\_name>**

**```**

This command pushes the local branch to the remote repository and sets it up as a tracking branch.

7. After successfully pushing your branch, it will be available in the remote GitHub repository for others to access and collaborate on.

Remember to replace `<remote\_repository\_url>`, `<remote\_branch\_name>`, and `<local\_branch\_name>` with the appropriate values specific to your repository and branch.

[alias]

s = status

co = checkout

br = branch

ci = commit