**Important git commands**

git restore file.name :used for reverting the changes or restoring to the last committed

state of the file

git restore –staged: restore the file the last committed state if the file has been staged

git log: for logging all the commit status

git log --oneline : reduced message size of the log message

git log --oneline --graph --decorate : To see the log in graphical manner . This is useful when you want to see the merging graph of different branches into master branch.

git status : give you status of tracked files that have been modified in the working directory

**configuring git**

local configuration:

git config user.name ="abc"

global configuration:

git config --global user.name= "abc"

so, whatever the commit happens form local machines it will have the name as 'abc'

**git branching**

git branch ---- will show all the branches

**creating new branch**-----------

git branch <newBranchname>

**move to newBranchname**----------

git checkout newBranchname

go back to the master branch--------

git checkout master

(this command will take you to the master branch)

single command to create a branch and switch to that branch------

git checkout - b <new-branch-name>

**Delete brach**

git branch -d <branch-name>

-------------------------

note **if a file has already been tracked with git add <filename> command**

**you can directly commit that file if you make any changes in the file**

with following command (called express commit)

git commit -a -m "message for the new commit "

this will add the file in staging area and then commit it .

--------**If you want to modify the last commit message or you want to add new file in the previous commit**------

git commit --ammend -m "this message will replace the last commit message and new file committed will also be added to the same last commit"

-----------------------

**Merging**

(Always come to the target branch for executing this command. In this case master branch)

if you want to merge the (newBranchname) into master branch

git merge <source-branch> , so source branch will be your newBranchname and that will be merged into

master branch provided you are in the master branch while executing this command.

**MERGING COULD BE OF TWO TYPE**

1. **FAST FORWARD MERGE** (normal merging with git merge <source-branch> where no conflict occures)

2. **CONFLICT MERGE** (Will create new commit in the branch that will keep the merge history)

let’s take an example

master branch is master, and another branch is newBranchname

currently another developer 'prashant' has checkedout newBranchname branch and let say he has modified the styles.css file .

At the same time some other developer 'rahul' is working in the master branch and modified the styles.css in the master branch.

now Prashant checks out the master branch and tries to merge the newBranchname into master branch then Prashant will get CONFLICT.

prashant will get a message saying Automatic merging failed; fix conflicts and then commit the result. **The status of the branch will be master| merging**.

prashant will manually have to resolve the conflict by opening the conflicting file and merging the conflicting changes from master as well as the newBranchname branch. finally, Prashant will have to commit the merge file again to change the status back to master branch from master | merging.

To avoid conflicting state or reverting the state of the conflicting file Prashant can run **'git merge --abort'** and merging will be reverted.

3**. REBASE BEFORE MERGING THEN COMPLETE MERGE WITH FAST FORWARD MERGE**

rebase can be used with merge or squash

**REBASE WITH SQUASH**:

squash is let say we have commits c1->c2->c3-c4 on same file , so instead we can have all these commits squashed into a single commit called squashed commit. This is nothing but rebase with squash.

**REBASE WITH MERGE:**

Consider an example main branch is master. master has series of commits as c1->c2-c3 (final commit is c3 hence head is pointing to c3). Another branch is created from masters latest commit ie c3. Let the name of the branch is newBranchname.

head pointer in newBranchname will point to c3. as the branch will be created from the latest commit c3.

now let say master has new commits after c3 as c1-c2-c3-c4 (head is pointing to c4 as the latest commit in master is c4).

lets say newBranchname also has new commits as c1-c2-c3-c11 (head is on c11 in newBranchname as the c11 is the latest commit in newBranchname).

if the newBranchname is to be merged in master branch and newBranchname is to be deleted then in rebase with merge following things are going to happen.

1. new commits in newBranchname ie c11 will be cached into memory.

2. new commits from the master branch id c4 will be commited onto c3 in newBranchname so head will point to c4 and commit will look like c1-c2-c3-c4

3. finally the cached commit of the newBranchname will be committed again onto c4. The commit will look like following c1-c2-c3-c4-c11 and head will point to c11

**Following commands are used to perform the above 3 operations**------

from newBranchname: 'git rebase master' (Remember conflict may come (that needs to be resolved manually and extra commit (first git add . , then git commit -m "appropriate messsage", if you want you can check status as well 'git status') will be required after resolving the conflicts finally run 'git rebase --continue' or 'git rebase --abort' to abort the rebase process altogether) while rebasing if same file has been modified in both master as well as the newBranchname branch or else rebase will happen without any conflicts)

4. Finally go back to master branch for merging the newBranchname branch with the master branch. Run command 'git checkout master'

and finally run 'git merge newBranchname' it will do the fast forward merge.

Basicaly by this approach all the conflicts will be resolved in the newBranchname branch. And finally the fast forward merge can happen in the master branch.

--------------------------------------------------------------------------------------------------------------------------------

Up until now we saw how to commit changes, now we are going to see how to undo some changes that has been committed wrongly.

**undo things -> working area, staging area, local repos(commit)**

Lets say we have only one branch ie master branch.

**Undo at WORKING AREA**---------

git restore <file-name> (will discard the changes in the working directory . state of the file will be restored to the last committed state)

**Undo at STAGING AREA** ------------

git restore --staged <file-name>

and if you want to again undo from the working directory run

git restore <file-name>

or above two can be done in one command using

reset are of two type soft reset and hard reset.

**soft reset (stage to unstage)**

'git reset <filename > 'will unstage the file from the staging area but the file changes will remain in the working directory.

**hard reset (unstage and delete)**

'git reset --hard <file-name>' it will unstage the changes from the staging area as well as undo the changes in the working directory and the state of the file will be restored to the last committed state.

if file name is not specified then all the files that has been staged will be unstaged and also undone in the working directory.

**UNDO Committed changes**---------

git reset and git revert can be used

git revert is decent version

git revert head === (revert changes done in last commit) after entering the message for the revert command and hitting enter the commit will be reverted to 2nd last committed state. and local directory will also be synchronized with the current committed state

git revert <id-of-the-commit> ===(revert changes done in that commit to its previous commited state)

git reset <id-of-the-commit>==(it will reset the head to id-of-the-commit and all the prior git history will be deleted). Hence if you do git status you will get files that needs to be committed ie the changes will be there in working directory but same will be deleted form remote and commit history will also be deleted.

but working directory will have those files or changes.

if git reset --hard <id-of-the-commit> is executed it will not only delete the commits and its history but also the changes the local directory will also be deleted (hence if you do git status it will tell you that the working directory is clean ie nothing to commit) so be careful about using this command.

This is very dangerous as it will delete the commit history permanently.

**-------------------checkout certain committed state-------**

lets say one the committed state id is we2311 in master branch and lastest committed id is er45

you did 'git checkout we2311' head will temporarily move from er45 to we2311 in master branch to this state id and your working directory will also have file from we2311 commit only.you can make chages in this state and commit new changes. Any new commit you do will be above we2311 commit.

But if you switch back to master branch by using git checkout master then all the commits done above we2311 will be lost .

If you want to retain the commits above we2311 then you have to create another branch for storing the commit you did above we3211 before switching back to master branch.

so after committing above we2311 run git switch -c <new-branch-name>.

Then finally you can switch back to master branch and the changes done above we2311 will be available in new branch that you created.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**git stash------**

when stash is used ?

let say you are working on a branch 'feature' and you have changes in file style.css and staged it and you have made chages in file app.js but you have not staged it . Now your tech lead says that you have to work on master branch

On ungent basis. You need to save your changes that are staged and not statged both.

In this case git stash command comes into picture.

If you run

git stash ====This will save the changes made to a file that have been tracked. ie styles.css;

git stash -u -m "some message you wanna add" ======this will save the changes made to a file that have not been tracked like app.js.

-u is for untracked files to be stashed.

-m "some message you wanna add" is optional in the command.

if you run git status you will find that the working directory is clean ie your changes have been stashed successfully!

----see the stashed list by running 'git stash list'

So once you have finshed your work in the master branch you can come back to feature branch and run following commands to restore the saved changes.

1. You can either run git 'stash pop stash{0}' (stash{0} is the id of the stashed file that can be found by running git stash list)This will bring the status of the file back to the way it was before git stash command ran.Entry of the stash{0} will be removed from the stash list.

2. Or you can run git 'stash apply stash{0}' it will do exactly what the first command did but the entry in the stash list will not be deleted it will still be present in the list to check you can run git stash list but if you try to reapply the same stash by running 'git stash apply stash{0}' then you will not be allowed to as you have already applied the stash.You can delete the stash list by running 'git stash drop'

Remote and local connection

You already know how to connect your local to remote and vice versa

Tips !

1. always do pull first before pushing to the server.

2. difference between git fetch and git pull

git fetch updates the local git repo. git pull updates the local git repo as well as it pulls the changes in the working directory as well.

--allow-unrelated-histories