GIT NOTES



**git hash-object**

**🡪** Creates a blob

**git mktree**

**🡪** Creates a tree

**find .git/objects -type f**

**🡪** Shows all the objects inside of the object folder.

**Git -file <hash> -t** (type of file) **or -s** (size of file) **or -p**(for the object of file)

🡪 Gets information about the files.

**git -files**

🡪 List all files in the staging area.

🡪 Adding -s to end of command, makes you get the files in table format.

**Output with -s:** 100644 b7aec520dec0a7516c18eb4c68b64ae1eb9b5a5e 0 file1.txt

0 shows that there are no differences between repository and staging are for that file.

**git read-tree <hash>**

🡪 Retrieves file from git repository and put them into the staging area.

🡪 It is going to try find the object name with hash we gave above,

If object is found it will take entire tree under the object to the

staging area.

**git checkout-index -a**

**🡪** Takes the file from staging are to the working directory.

**🡪** Without -a it only reads files in staging area.

**git cat-file <-p, -s, -t>**

**🡪** With different options it reads contents and information about the objects from git database. (-p , -s)

**cat <filename>**

**🡪**  Shows the content of the file.

**git status**

**🡪** Show the status of the repo

**git config –global user.name <Name>**

**git config –global user.email <Email>**

**🡪** Allows us to set the name and email info to the repo

without -global it just overrides only the current repor informations

**git config --list**

**🡪** Show the list of configs

**git add**

**🡪** Add files to staging area

**git commit**

**🡪** Write changes to Git repository

**git log**

🡪 History of changes (commits)

**git checkout <commit(hash of commit)> or <branch>**

**🡪** Checkout commmit or branch (makes you travel between different versions of the project) (from repository to working directory) (moves the HEADER to specific commit or branch)

**🡪** It changes the reference of HEAD to different branch or commit

**git rm –cached <filename>**

**🡪** Makes the status of file from Staged to Untracked

B**ranches**

**🡪** If branch is not merged and want to be deleted it must be typed -D instead of -d

**git checkout -b <branch name>**

**🡪** Shortcut for creating a branch with checkout

A close-up of a list of text

Description automatically generated

**git diff**

**🡪**  Shows the previous and current version of the repo. (a file = previous version, b file = new version)

**Merging Branches**

**-> Fast forward merge**

Fast forward merge is possible only when there are no further commits in the receiving branch ( X branch is called receiving branch, If we are merging from y to x branch.) after the commit where y branch was created.

If we want to merge into main branch or another branch we should checkout to this exact branch first.

A diagram of a process

Description automatically generated



After Merging with Fast Forwardard

A diagram of a process

Description automatically generated

After merging we can delete feature branch with ‘ git branch -d <branch\_name> ’

A screenshot of a computer

Description automatically generated

* Git will automatically decide which merging technique is going to be used.

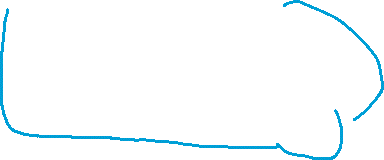
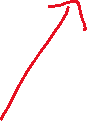
**-> Three way merge**

A diagram of a diagram

Description automatically generated

A diagram of a timeline

Description automatically generated



After merging feature1 into master we can safely delete feature1. When feature1 is deleted it’s commmits are NOT going to be deletedes as above.

**Merge Conflict**

Merge conflict appers when same file is edited in different branches. You must resolve the conflict manually. You should resolve it from master or sender branch and commit them. After this, git is going to merge branches automatically.

It is not possible to have merge conflict in fast forward merge.

After resolving conflict we simply add them to staging area and commit changes without any commit message.

**Notes:**

* When the HEAD is attached to any commit but not last commit it is in Detached Head State.

**GitHub**

A diagram of a remote git reposition

Description automatically generated

**git clone <https link>**

**🡪** Clones to repo from remote to local.

**git fetch**

**🡪** Updates the local repo according to remote repo. It doesn’t effect staging area and working directory.

**git pull**

**🡪** Equals to **git fetch + git checkout**. Write changes to staging area and working directory.

Merges remote branch into current branch

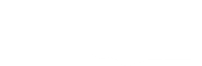
\*\*This command only updates only single local currently checked out branch

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



**git pull -v**

A computer screen shot of a computer program

Description automatically generated

**FETCH\_HEAD**

**🡪** the sha1 hash without tag ‘not-for-merge’ is the very last commit of the romote repository

**git merge FETCH\_HEAD <branch name>**

**git remote -v**

**🡪** Shows the remote servers of the repo.

**git remote show origin**

**🡪** Shows the entire information about connection between local and remote repository.

**git branch -a**

**🡪** Shows the all branches in local and remote repo

with -vv option intead of -a we get local branches and their corresponding remote branches

with -r option we see remote branches

**git fetch**

**🡪** Updates the branches in local according to remote

**git push -v**

**🡪** s

**git push --set origin <remote branch name> OR git push -u origin <remote branch name>**

**🡪** If new branch created in local and absent in remote repository, we must create first branch in the remote and then with ‘push -v’ option push the changes.

**git push origin -d <remote branch >**

**🡪** If you want to remove branch from local and remote at the same time you should use this command.

**git show-ref**

**🡪** Shows the local and remote references.

**🡪** If you add name of the branch at the end of the command you will see the references for the written brach. Thus, you can easily compare them whether these are in the same commit or not.

**git commit –amend –author=”<name of the user> <<email of the user>> ”**

**🡪** Changes the last commit’s author information in local repository and keep the author informations for upcoming commits

**Deleting branch from remote while it still exist in local**

A screenshot of a white text

Description automatically generated

**git remote add <name of the repo is going to be created> <HTTPS link of remote repo>**

**🡪** If you have forked repo and you want to pull changes from parent repo of forked repo, you must add remote server to your local repo of the parent of forked repo.

A screen shot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

**git pull <remote serve name> <local branch name> - v**

**🡪** We sync changes from parent repo to local repo. After this command we can push changes to our forked repository.

**Notes:**

* “tree” in the GitHub URL doesn’t represent Git tree object. Instead it is just pointer to specific branch or commit.

/tree/master

/tree/6524dfb2

* after cloning remote repository although there is multiple branches in the remote repository, git creates only one branch with the name of default remote branch.
* we simply “**git checkout <remote branch name>**” to get the remote branch which is present in remote but absent in local.(actually is it also present in local but in the Git repository. So we must chechkout If we want to create corresponding tracking branch of remote branch. If it doesn’t even exist in Git repository when must “**git fetch**” for getting latest informations from remote repository)

**Pull Request**

**FORK**

A white text on a white background

Description automatically generated

**GIT TAG**

* Git tags are not pushed to remote with “git push"
* Annotated tags are stored as a object unlike lightweight

**git tag**

**🡪** Show the tags in the repo

**git push –tags**

**🡪** Pushes the tags created locally in the repository

**git push -v <remote server name> <name of the tag>**

**🡪** This command also pushes the tag to remote. This command and command above pushes only tags not other changes

**git tag <name of the tag>**

**🡪** Creates a lightweight tag with the given name.

**git tag -a <name of the tag> -m “tag message”**

**🡪** Createsa annotated tag with the given name and message

A computer screen with text and numbers

Description automatically generated

**git tag -v <existing tag name>**

**🡪** Show the details of the annotated tag

**🡪** Does not show the lightweight tags because they are not even git object

**git show <name of the tag>**

**🡪** Show the infos of the tag (in which commit etc.)

A computer screen shot of text

Description automatically generated

**REBASING**

A screenshot of a computer screen

Description automatically generated

A diagram of a tree

Description automatically generatedA diagram of a process

Description automatically generated

**GIT IGNORE**

**🡪** If you want to ignore all files inside the folder you can simpy write into .gitognore file like this:

your\_folder\_name/

**🡪** You can add comments into .gitgnore files with #.

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

This above command is going to remove the file git repository but keep it in workin directory

A screen shot of a computer

Description automatically generated

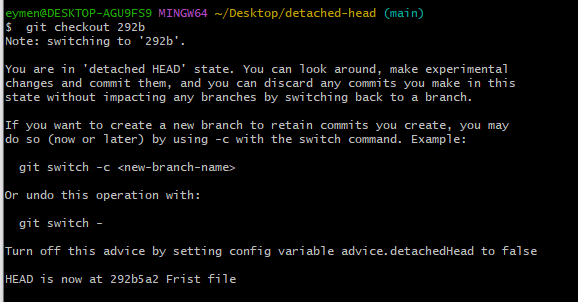
A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

**DETACHED HEAD STATE**



**🡪** After checking back to main bracnch, commits we have made in detached head state ara going to be gargabe collected If we did not create a branch from them.

**🡪** Git objects are garbage collected after 30 days.

🡪 You can also create a branch from detached head state with the commad below

**git checkout -b <new branch name>**

A computer screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**ADVANCED GIT**

**GIT LOG**

**git log –oneline**

**🡪** Show the logs in one line

**git lg**

**🡪** Similar to one line but it contains author informations etc.

**git log –graph**

**🡪** Show the logs with a graphic

🡪 If we add –oneline option to end of this command we can see the commits which has multiple commits.

**git log –stat**

**🡪** Shows the quantity of actual changes made in the commit.

**git log -p**

**🡪** Show the every change inside of the all filen in the commit.

**git log -<number of commits>**

**🡪** Shows the exact number of commits from starting th end.

**git shortlog**

**🡪** Show the logs of the authors.

**git shortlog -n**

**🡪** Sorts according to quantity of commits by every author.

**git shortlog -n -s**

**🡪** it shows in shorter format

**git shortlog -n -s -e**

**🡪** Shows also emails

**gitt log –author==”<author name>”**

🡪 Show us to commits done by the exact author

🡪 We can also add –oneline option to end of the command.

**git log –grep=”<text for query>”**

**🡪** Shows us the commits which commits message contains the query text.

**git log –pretty=format:”%H”**

**🡪** Show only SHA1 hash of all commits

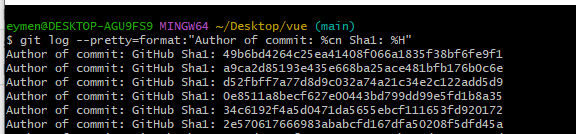
🡪 With the lower h instead of H it shows the shortened hash.

* for more formatting variables: https://devhints.io/git-log-format

**git log –pretty=format:”%cn %H”**

**🡪** Shows SHA1 hash and author of all commits

🡪 cd = commit date

\

**git log --merges**

**🡪** Shows the merge commits .

**git log --no-merges**

**🡪** Shows the all commits except merge commits.(only human made commits)

**GIT RESET**

**🡪 Git reset is a destructive command.**

A blue rectangular sign with white text

Description automatically generated

**git reset –soft <hash>**

**🡪** After resetting, it;

* Discards commits
* Keep changes in staging area
* Keep changes in working directory

🡪 After this command you can again commit staging are to repo.

**git reset –mixed <hash> (default of git)**

**🡪** After resetting, it;

* Discards commits
* Discard changes in staging area
* Keep changes in working directory

🡪 After this command you can add the changes preserved in working directory to staging area and commit them.

**git reset –hard <hash>**

**🡪** After resetting, it;

* Discards commits
* Discards changes in staging area
* Discards changes in working directory

🡪 After this command there is no going back.

**git reset HEAD~5**

**🡪** Effects last 5 commits

**GIT REVERT**

* With revert we can only revert one commit.
* It is not destructive operation unlike git reset.

**git revert HEAD**

**🡪** Creates a revert commit in he current sha1 hash of the HEAD.

🡪 This command creates a new commit which has reverted changes.

FROM

A screenshot of a computer

Description automatically generated

TO

A screenshot of a computer

Description automatically generated

**GIT AMEND**

* You can only use this very last commit

**git commit -amend -m “<new desired message of the commit>”**

**🡪** It does not create a new commit. However, it changes the SHA1 Hash of the commit changed

🡪 It only effects the very last commit

**git commit –amend –author=”<name of the author> <<mail of the author>>”**

**🡪** Changes the author informations

**CHERRY PICK**

* Allow us to take any one commit from repo and insert into currently checkout branch as a last commit.

**Scenraio**

**🡪** New branch created and we made some several commits.

🡪 After this we do not want to merge it but anly take the commit from there and insert it into main branch.

**git cherry-pick <hash>**

**🡪** Takes the commit and inserts it into currently checkedout branch.

🡪 This command creates a new commit while inserting,

**git cherry-pick –no-commit <hash>**

**🡪** Unlike the above command it does not commit but only stages. (Allows us to commit with different messages)

**REFLOG**

* Allows us checkout into specific commit from the history.
* Operations in the reflog are stored by default 90 days.
* For example, after hard reset we made. We can go back to old version wit hard reset of of the commit from the reflog history.(With hard reset removed changes are going to be garbage collected 30 days later. Thus, we can go back to previous commit with hard reset again.)

A computer screen shot of text

Description automatically generated

A computer screen shot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

**git reflog**

* This command shows the entire history of all operations made in repository.
* This command outputs only operations made in **local** repository.

A screenshot of a computer program

Description automatically generated

**git reflog show <branch name>**

* Shows the all operations made in the exact branch.

**STASHING**

* Allows user to save uncommitted changes.

**Scenario**

* You are workin on the temp1 branch and you have changes in working directory and staging are in this branch. After you are done with temp1, you want to checkout to temp2 branch but at the same time you don’t want to loose anything in temp1. Thus you stash changes in temp1. After you come to temp1 from temp2 you can retrieve stasched changes and continue your work.

**git stash**

**🡪** After this command, git creates a temporary commit, stores the stashed changes in git repository. And it creates reference to the temp commit insided of the .git/refs/stash

**git stash pop**

**🡪** After git stash we pop them If we want to use it. After this commnad, git deletes the .git/refs/stash file.

in github desktop

A screenshot of a computer

Description automatically generated

**GARBAGE COLLEECTION**

* Garbage collector will delete absolute records, and packs git objects in a pack file. (Pack file is archive of git objects that allows us to make git repository much more smaller.)
* removes unreachable objects, old reflog records.

A computer screen shot of a program

Description automatically generated

**REBASING WITH SQUASHING**

* Thir merge process takes all the commits made in the feature1 branch and puts it into one commit with appending their commit message into one commit message unlike normal rebasing.

A screenshot of a computer screen

Description automatically generated

**in local repo**

**git rebase -i <sha1 hash of the commit>**

A computer screen shot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

If you want to rebase with stashing you must write s after deleting pick except first line.

In this case git replaced three commit with one commit inside of the feature2 branch.

After stashing we can safely merge feature2 into main branch. So, stashing rebase process in local is done.

**GIT HOOKS**

* Allows us to create a condition for event of the git (commit, precommit etc.) in only local repo.
* In order to give a execution permission to hook:
* **chmod +x <hook name>**
* You can bypass all hooks with **–no-verify** option.

A screenshot of a computer screen

Description automatically generated

With the ‘sample’ extension, git disables all the hooks as a default feature. If you want to enable any of them, you simply need the remove the extension like from **applypatch-msg.sample** TO **applypatch-msg.**

The above files are exceuteable files because their permissions “-rwxr-xr-x” contains “r”.

**pre-commit hook**

**-** The below pre commit hook is executes 1 (which is unsuccessful) for all cases. It does not even check anyting.

A screenshot of a computer program

Description automatically generated

This pre-commit hook checks the user email whether it fits to rule or not.

A screenshot of a computer program

Description automatically generated

**post-commit hook**

**-** It starts after committing.

A screen shot of a computer

Description automatically generated

**pre-rebase hook**

**-**Prevents rebasing

A screen shot of a computer

Description automatically generated