**Frequently used git commands**

1. **Git config**: sets author name & email.

Git config –global user.name “username”

Git config –global user.email “email”

1. **Git clone**: to obtain a repository from an existing url.
2. **Git clone –b <branch name> url:** to obtain particular branches.
3. **Git commit**: records or snapshots the file permanently in the version history.

**git commit –m :** changes the head and records or snapshots with a message.

**Git commit –a :** commits any files added in the repository with git also commits any files changed since then

**Git commit –amend:** it will prompt the default text editor and allow us to edit the commit message.

1. **Git pull:** git pull [repository link]

Fetches and changes on the remote server to your working directory.

1. **git push**: git push [variable name]master

Sends the committed changes of master branch to your remote repository.

                : git push [variable name][branch]

Sends the branch commits to your remote repository.

                 : git push –all [variable name]

Pushes all branches to your remote repository.

                : git push [variable name]:[branch name]

Deletes a branch on your remote repository.

1. **Git stash save:** temporarily stores all the modified tracked files.
2. **Git stash pop:** restores most recently stashed files.
3. **Git stash list:** list all stashed changesets.
4. **Git stash drop:** discard the most recently changeset.
5. **Git revert:** to undo the changes.

**git commands**

1. **Git –version**: to know the version of the application.
2. **Git config**: sets author name & email.

Git config –global user.name “username”

Git config –global user.email “email”

1. **Mkdir**: used to create a new repository.
2. **Cd:** command used to change directory.
3. **Git init**: used to create a blank repository.

It is used to make an existing project as a Git project. Several Git commands run inside the repository, but init command can be run outside of the repository.

1. **Git clone**: to obtain a repository from an existing url.
2. **Git add**: add a file to the staging area.

Git add [file]

Git add\* adds one or more to the staging area.

1. **Touch**: used to create new empty file.
2. **Ls :** lists the files in the directory.
3. **Git commit**: records or snapshots the file permanently in the version history.

**git commit –m :** changes the head and records or snapshots with a message.

**Git commit –a :** commits any files added in the repository with git also commits any files changed since then

**Git commit –amend:** it will prompt the default text editor and allow us to edit the commit message.

1. **Git diff**: show the file differences which are not yet staged.

**Git diff –staged:** shows difference between the two branches mentioned.

1. **Git reset**: unstages the file, but it preserves the file contents.

**git reset [commit]:** undergoes all the commits after the specified commit& preserves the changes locally.

**Git reset –hard[commit]:** discards all history and go back to the specified commit.

1. **Git status:** lists all the files that you have to be committed.
2. **Git rm:** deletes the files from your working directory & stages the deletion.
3. **Git mv [existing path] [new path]**: change an existing file path and stage the move.
4. **Git log:** used to list the version history for the current branch.

**Git log –follow[file]:** lists version history for a file, including renaming of files also.

1. **Git show:** shows metadata and content changes of the specified commit.
2. **Git tag:** to gives tags to the specified commit.
3. **Git branch:** lists all the local branches in the current repository.

**Git branch [branch name]:** creates a new branch

**Git branch –d [branch name]:** deletes the feature branch.

1. **Git checkout:** used to switch one branch to another.

**Git checkout –b[branch name]:** creates a new branch also switches to it.

1. **Git merge**: merges the specified branch history into the current branch.
2. **Git remote:** used to connect to your local repository to the remote server.
3. **git push**: git push [variable name]master

sends the committed changes of master branch to your remote repository.

                : git push [variable name][branch]

sends the branch commits to your remote repository.

                 : git push –all [variable name]

pushes all branches to your remote depository.

                : git push [variable name]:[branch name]

deletes a branch on your remote repository.

1. **Git pull:** git pull [repository link]

fetches and changes on the remote server to your working directory.

1. **Git stash save:** temporarily stores all the modified tracked files.
2. **Git stash pop:** restores most recently stashed files.
3. **Git stash list:** list all stashed changesets.
4. **Git stash drop:** discard the most recently changeset.

**Git pull**

The term pull is used to receive data from github. It fetches and merges changes from the remote server to your working directory. The git pull command is used to pull a repository.

**Git pull <option> [<repository url><refspec>..]**

**<option>:** options are the commands. Options can be –q(quiet), -v(verbose), -e(edit) and more.

**<repository url>:** git hub or any git service urls.

**<refspec>:** a ref is referred to commit, for example, head (branches), tags and remote branches.

**Pull request:** Pull request allows you to announce a change made by you in the branch.

Pull requests **let you tell others about changes you've pushed to a branch in a repository on GitHub**. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

**Git push**

The push term refers to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository.

**Git push <option> [<remote url><branch name><refspec>..]**

**Git push tags:**

**<repository>:** the repository is the destination of a push operation. It can be either a url or the name of a remote repository.

**<refspec>:** it specifies the destination ref to update source object.

**--all:** the word “all” stands for all braches. It pushes all branches.

**--prune:** It removes the remote branches that do not have a local counterpart.

**--mirror:** It is used to mirror the repository to the remote.

**--dry-run:** dry run tests the commands. It does all this except originally updates the repository.

**--tags:** it pushes all local tags.

**--delete:** it delete the specified branch.

**-u:** it creates an upstream tracking connection. It is very useful if you are going to push the branch for the first time.

**Git stash**

Sometimes you want to switches the branches, but you are working on an incomplete part of your current project. You don’t want to make a commit of half-done work. Git stashing allows you to do so.

Git stash command enables you to switch branches without committing concerning repository and working directory.

Meaning of stash is **store something safely in a hidden place.**

**Git status:** to check the current status of the repository.

**Git stash:** to save untracked files temporarily.

**Git stash save “<stashing message>”:** to stash a change with a message.

**Git stash list:** to check stored stashes.

**Git stash apply:** you can re-apply the changes that you just stashed by using the git stash command.

**Git stash apply <stash id>:** In case of more than one stash, you can use “git stash apply” command followed by stash index id to apply the particular commit.

**Git stash show:** shows the file that is stashed and changes made on them.

**Git stash show –p:** -p stands for partial stash. It shows the edited files and content.

**Git stash pop:** it will re-apply the previous commits to the repository.

**Git stash drop:** is used to delete a stash from the queue. Generally it deletes more recent stash.

**Git stash drop <stash id>:** we can delete particular stash from the queue.

**Git stash clear:** this command allows deleting all the available stashes at once.

**Git stash branch <branch name>:** the above command will create a new branch and transfer the stashed work on chat.

**Git revert**

In git, the term revert in used to revert some changes. The git revert command is used to apply revert operation. It is an undo type command. It does not delete ant data in this process.

It can be useful for tracking bugs in the project. If you want remove something from history the git revert is a wrong choice. Git revert records some new changes that are just opposite to previously made commits.

**Git revert:** to undo the changes.

<commit>: commit option is used to revert a commit. To revert a commit, we need the commit reference id.

**Git revert <commit-ish>**

**< --edit>** it is used to edit the commit message before reverting the commit.

**Git revert –e <commit-ish>**

**-m parent-number /--mainline parent-number:** it is used to revert the merging.

**-n/--no edit:** this option will not open a text editor. It will directly revert the last commit.

**Git revert –n <commit-ish>**

**--cleanup=<mode>:** it determines how to strip spaces and comments from message.

**-n/--no-commit**: generally, the revert command commits by default. The no commit option will not automatically commit.

**Git revert to previous commit**

Suppose we have made a change to a file say new.txt of your project. And later, you remind that you have made a wrong commit in the wrong file or wrong branch. Now we want to undo the changes.

**Git log:** it will show most recent commit-ish.

Copy that code and then run the command

**Git revert (code):** then changes made on the repository have been reverted.

**Git revert merge**

Again git log command will display the commit history. Copy the merging commit

Git revert <commit reference> -m1:

The above command will revert the merging operation.

**HOTFIX BRANCH**

**H**otfix branches are created to fix specific bugs in production environment which were introduced after deploying previous release branches.

When working with hotfix branches, you should open up a “pull request” in github so that you team members can see what you are preparing to fix. This is considered as the best practice.

**PREPARATION:**

These steps are compulsory before start working on a new hotfix branch because our local branch might be behind remote copy.

1. Make sure we are on develop branch.

Git branch

1. Fetch all remote updates.

Git remote update

1. Update local develop branch so it is up-to-date with remote copy.

Git pull origin develop

1. Checkout to master branch.

Git checkout master

1. Update local master so it is up-to-date with remote copy.

Git pull origin master

**CREATE HOTFIX BRANCH**

1. Check the current git status.

Git branch –avv

1. Create a hotfix branch that branches off of local master and tracks origin/master.

Git checkout –b hotfix/0.1.1 origin/master

Git branch

1. Fix the bug and commit to it.

Cat one.txt

1. Push hotfix branch to remote repository.

Git push origin hotfix/0.1.1

1. If you go to Github, there will be a notification bar that will ask you to open a new “pull request” for the hotfix branch you have just pushed. Open it by comparing it to master branch, writing a subject and a description for it.
2. At this point “peer review” takes place against the “pull request” in Github. If everyone in your team is happy with the work you have done in hotfix branch, there is nothing to worry about otherwise you keep working to make everyone happy. This process just shows everyone what will be fixed. Do not use “merge pull request” button.
3. Checkout into master branch.
4. Merge hotfix branch into master branch. Do not use –no-off flag otherwise merge will use “recursive” strategy instead of “fast-forward” and this will lead Github to create a pull request notification after pushing master branch to remote which is an unwanted behavior.
5. Tag your hotfix point by creating a new tag.
6. Verify that the tag has been created.
7. Push master branch to remote repository. In Github, this will also merge and close the “pull request” of remote hotfix branch into master branch however it won’t delete it. You will see that in Github. Also, you will pushing develop branch to remote repository later on.
8. Push the tags to remote repository. If you go to Github, you will go to Github, you will see the tag under “release” tab.
9. Checkout into develop branch.
10. Merge hotfix branch into develop branch. We did some work on file called 'one.txt' in develop branch after creating a release branch while ago so 'one.txt' in develop branch is more up-to-date than the one in master branch. We need to resolve conflicts manually and commit to it.
11. Push develop branch to remote repository.
12. Remove hotfix branch from the local repository.
13. Remove hotfix branch from the remote repository.
14. If you go to Github, you will should see cases below happened.

The master and develop braches are not even.

The hotfix tag “0.1.1” is there.

The hotfix branch itself is deleted.

The hotfix “pull requested” that we have opened is deleted.

**README.md file:**

README.md files are **Markdown files that describe a directory**. GitHub and Gitiles renders it when you browse the directory. For example, the file /README.md is rendered when you view the contents of the containing directory: https://github.com/google/styleguide/tree/gh-pages.