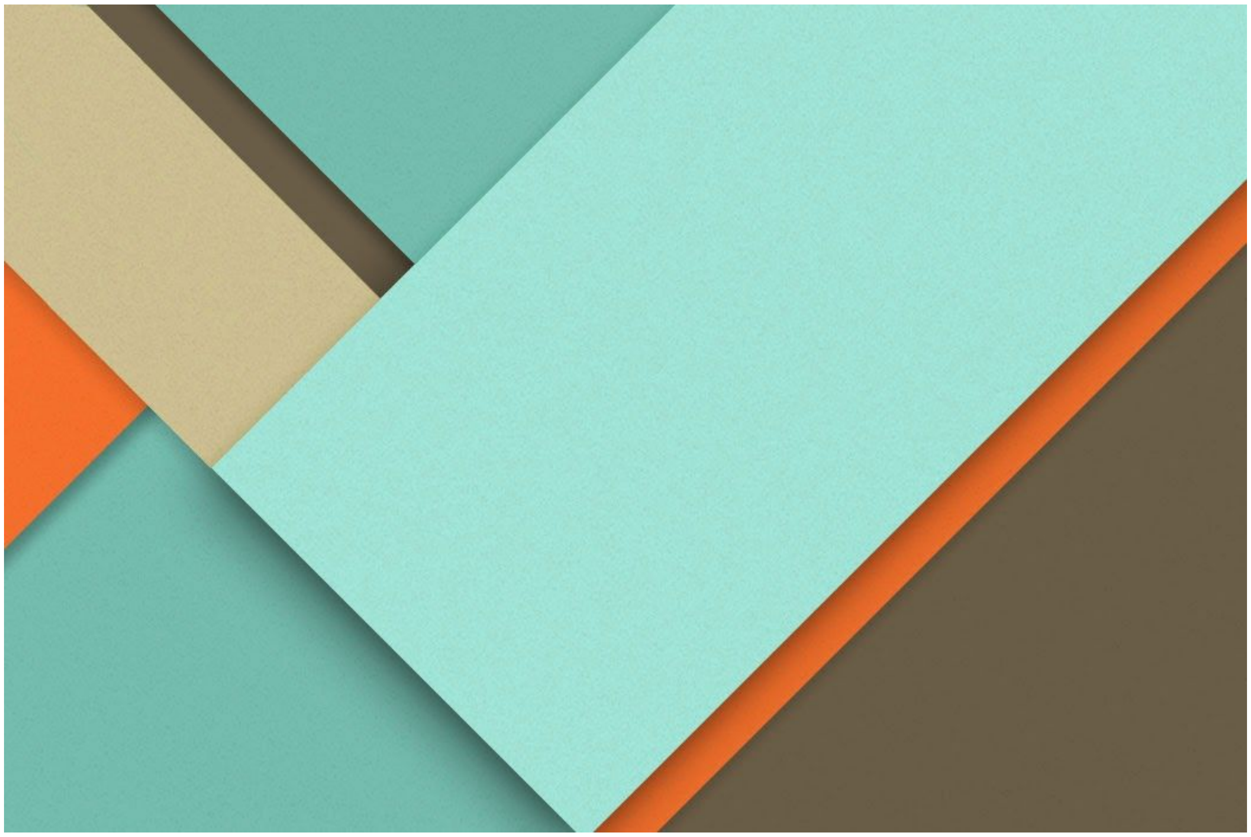


Git

Praveen Reddy Jirra
Senior DevOps Analyst

Overview

Git is a small yet very efficient version control tool. It helps both programmers and non-programmers keep track of the history of their project files by storing different versions of them.

What is Git?

Git helps developers keep track of the history of their code files by storing them in different versions on its own server repository, i.e., GitHub. Git has all the functionality, performance, security, and flexibility that most of the development teams and individual developers need.

Why Git Version Control?

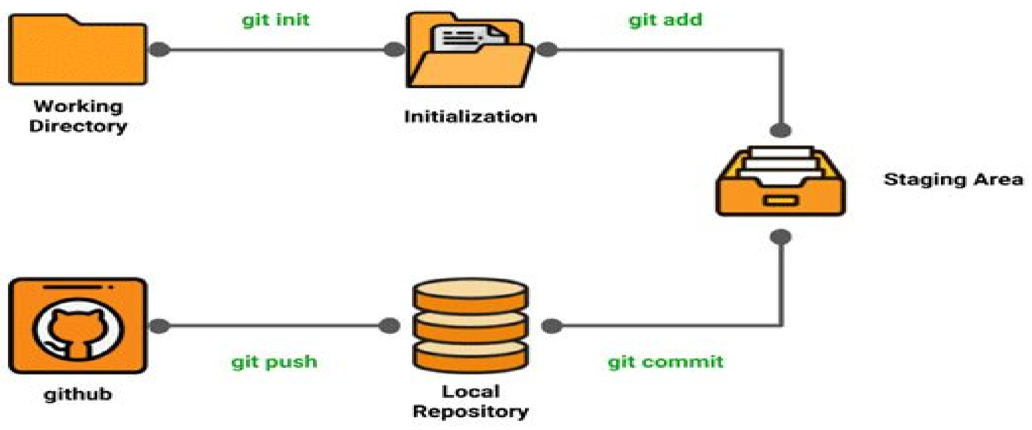
Below are some of the facts that make Git so popular:

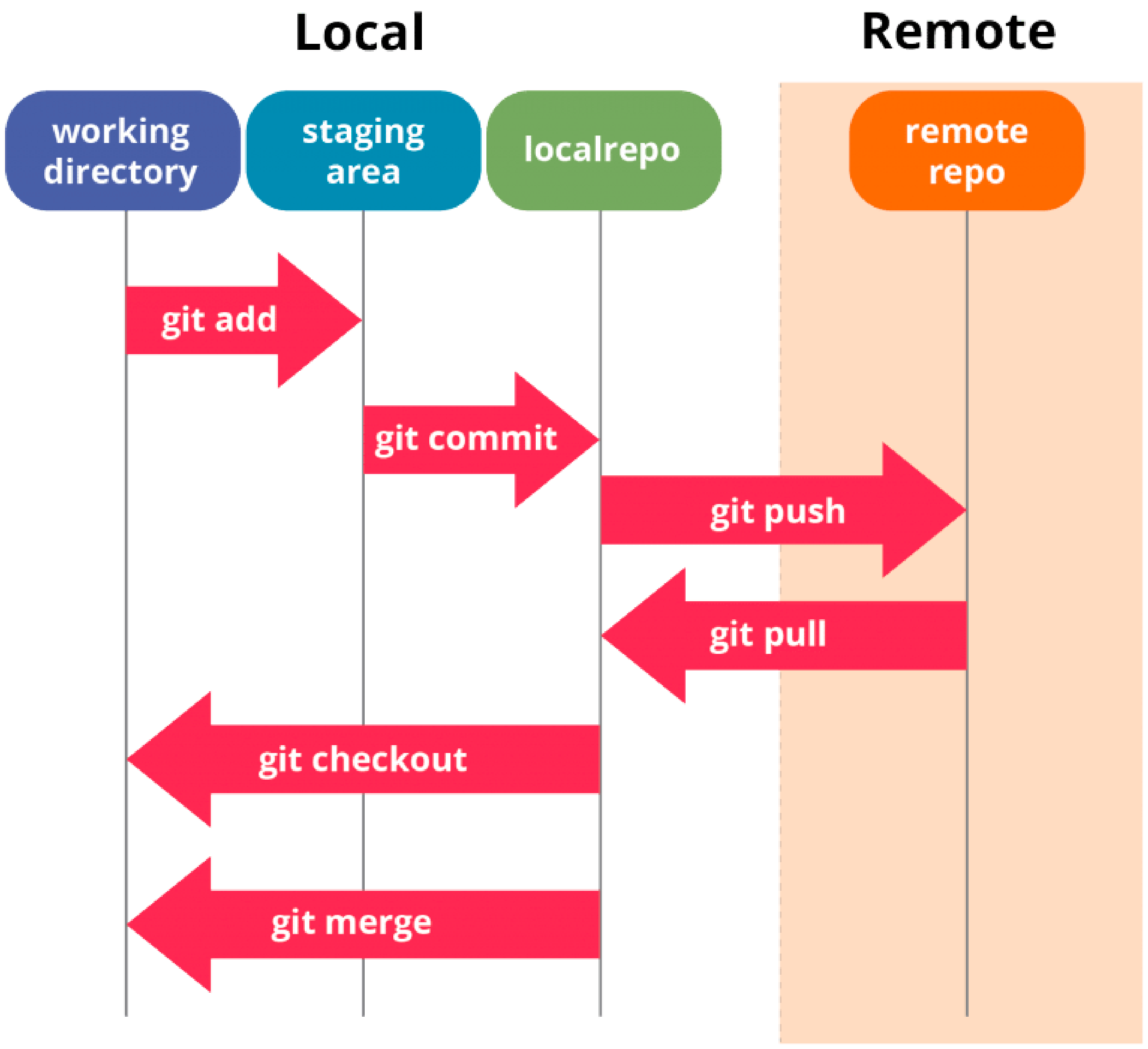
- Works offline: Git provides users very convenient options such as allowing them to work both online and offline. With other version control systems like SVN or CVS, users need to have access to the Internet to connect to the central repository.

- Undoes mistakes: Git allows us to undo our commands in almost every situation. We get to correct the last commit for a minor change, and also we can revert a whole commit for unnecessary changes.
- Restores the deleted commits: This feature is very helpful while dealing with large projects when we try out some experimental changes.
- Provides security: Git provides protection against secret alteration of any file and helps maintain an authentic content history of the source file.
- Guarantees performance: Being a distributed version control system, it has an optimized performance due to its features like committing new changes, branching, merging, comparing past versions of the source file, etc.
- Offers flexibility: Git supports different nonlinear development workflows, for both small and large projects.

Git Life Cycle

- Local working directory: The first stage of a Git project life cycle is the local working directory where our project resides, which may or may not be tracked.





- Initialization: To initialize a repository, we give the command git init. With this command, we will make Git aware of the project file in our repository.
- Staging area: Now that our source code files, data files, and configuration files are being tracked by Git, we will add the files that we want to commit to the staging area by the git add command. This process can

also be called indexing. The index consists of files added to the staging area.

- Commit: Now, we will commit our files using the git commit -m ‘our message’ command.

We have successfully committed our files to the local repository. But how does it help in our projects? The answer is, when we need to collaborative projects, files may have to be shared with our team members.

This is when the next stage of the Git life cycle occurs, i.e., in GitHub, we publish our files from the local repository to the remote repository. And how do we do that? We do that by using the git push command.

- **git init** initializes a brand new Git repository and begins tracking an existing directory. It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control.
- **git clone** creates a local copy of a project that already exists remotely. The clone includes all the project’s files, history, and branches.
- **git add** stages a change. Git tracks changes to a developer’s codebase, but it’s necessary to stage and take a snapshot of the changes to include them in the project’s history. This command performs staging, the first part of that two-step process. Any changes that are staged will become a part of the next snapshot and a part of the project’s history. Staging and committing separately gives developers complete control over the history of their project without changing how they code and work.
- **git commit** saves the snapshot to the project history and completes the change-tracking process. In short, a commit functions like taking a

photo. Anything that's been staged with git add will become a part of the snapshot with git commit.

- **git status** shows the status of changes as untracked, modified, or staged.
- **git branch** shows the branches being worked on locally.
- **git merge** merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature branch into the master branch for deployment.
- **git pull** updates the local line of development with updates from its remote counterpart. Developers use this command if a teammate has made commits to a branch on a remote, and they would like to reflect those changes in their local environment.
- **git push** updates the remote repository with any commits made locally to a branch.

Git Practical:-

Before getting into the repositories and all, we first need to configure user information for all the local repositories. Follow along :)

```
$ git config --global user.name "praveenjirra"
$ git config --global user.email "praveen.jirra@gmail.com"
$ git config --global color.ui auto
```

user.name — Sets the name you want attached to your commit transactions

user.email — Sets the email you want attached to your commit transactions

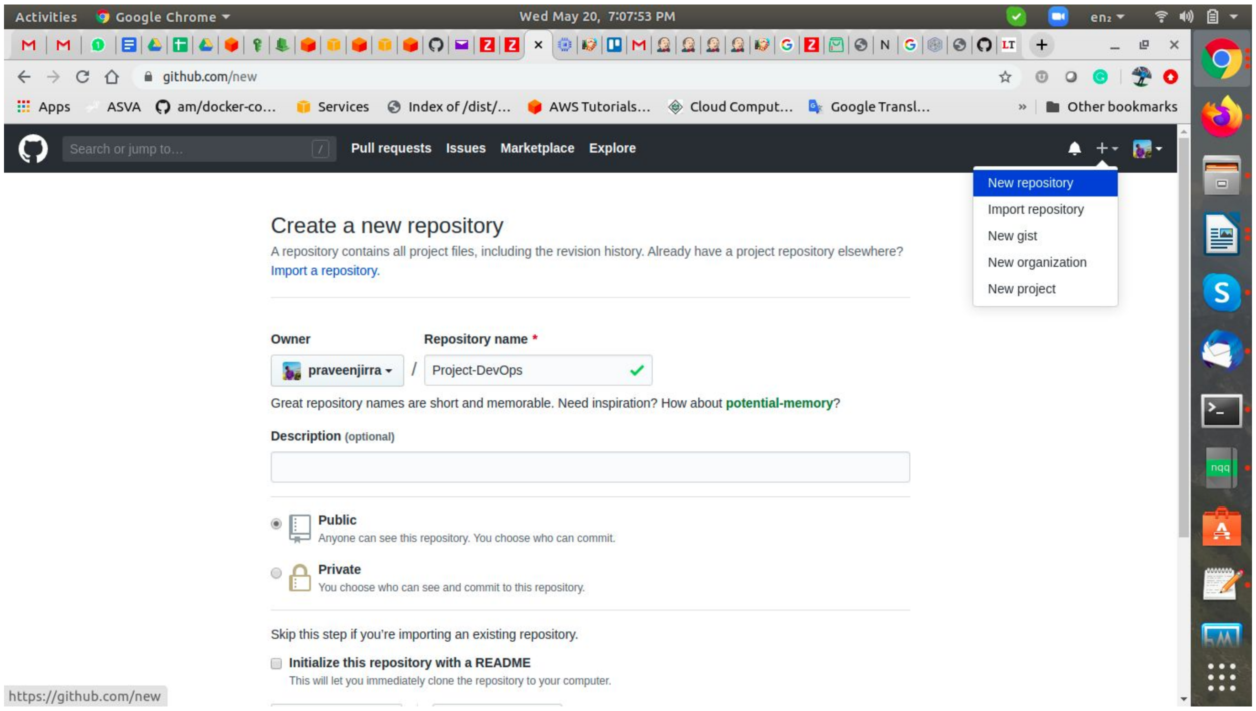
color.ui — Enables helpful colorization of command line output

Git Installation:

$ sudo apt-get install git

$ git --version

Create Repo:-



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner: praveenjirra / Repository name *: Project-DevOps

Great repository names are short and memorable. Need inspiration? How about **potential-memory**?

Description (optional)

(●) **Public**
Anyone can see this repository. You choose who can commit.

( ) **Private**
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

[ ] **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

> git clone https://github.com/praveenjirra/Project-DevOps.git

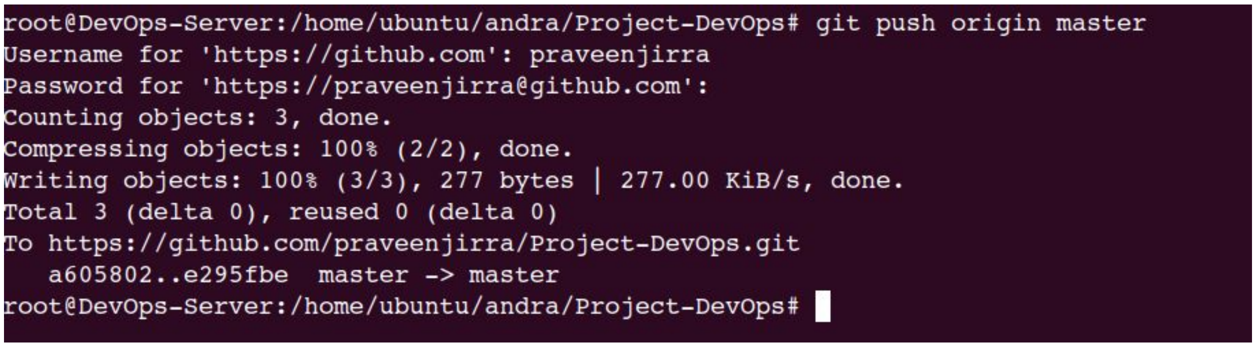
> cd Project-DevOps/

> touch git-demo.txt

> git add git-demo.txt

> git commit -m "first-commit"

> git push origin master



```
root@DevOps-Server:/home/ubuntu/andra/Project-DevOps# git push origin master
Username for 'https://github.com': praveenjirra
Password for 'https://praveenjirra@github.com':
Counting objects: 3, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 277 bytes | 277.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/praveenjirra/Project-DevOps.git
   a605802..e295fbe  master -> master
root@DevOps-Server:/home/ubuntu/andra/Project-DevOps#
```

Branch Creation:

Method-1

> git branch Dev

> git checkout Dev

> git add git-demo1.txt

> git commit -m "new-branch"

> git push origin Dev

Method-2

> git checkout -b Prod

> git checkout Prod

> git add git-demo2.txt

> git commit -m "new-branch"

> git push origin Prod

Git Merge:-

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.

> git checkout -b praveen master

> git branch

> touch merge.txt

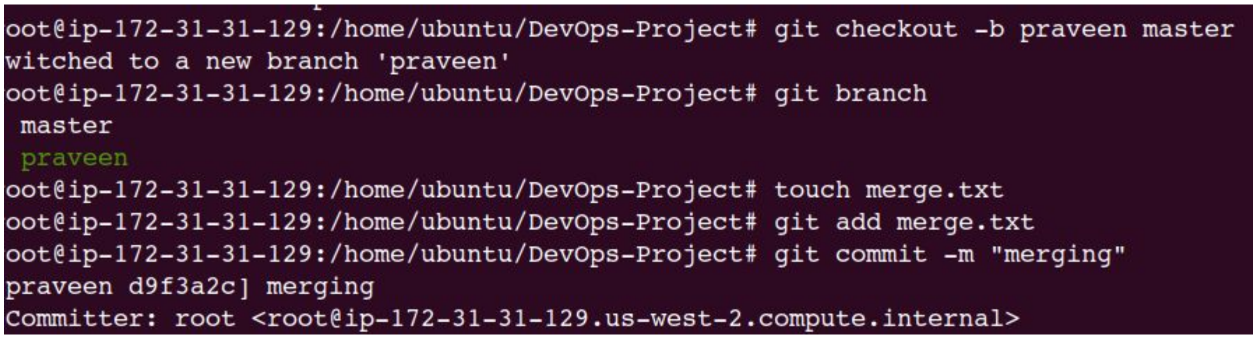
> git add merge.txt

> git commit -m "merging"

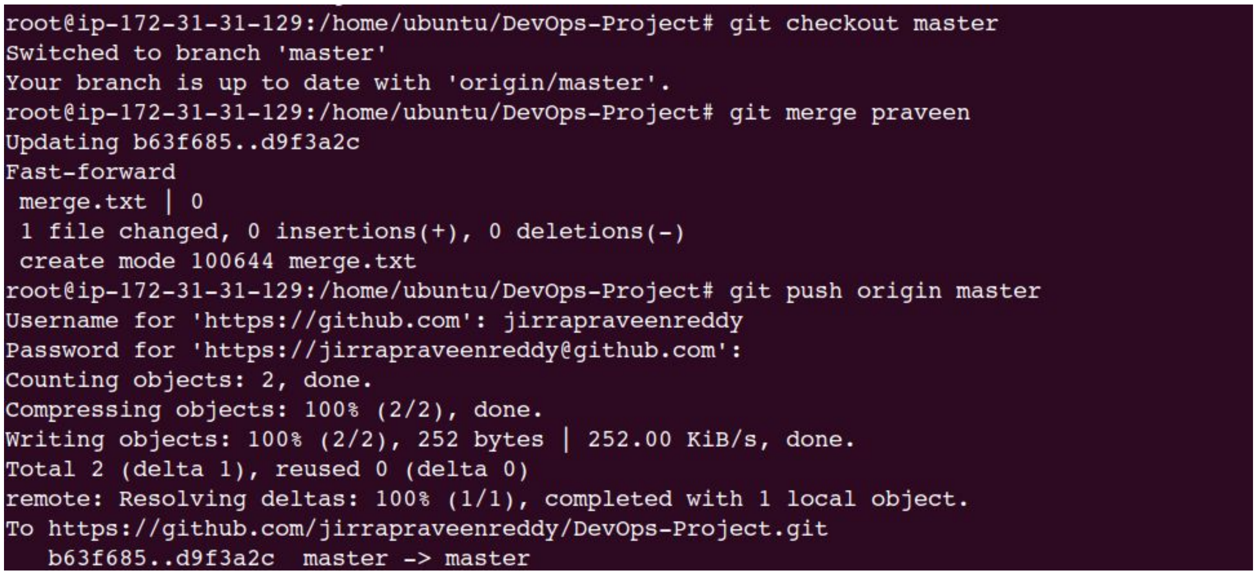
> git checkout master

> git merge praveen

> git push origin master



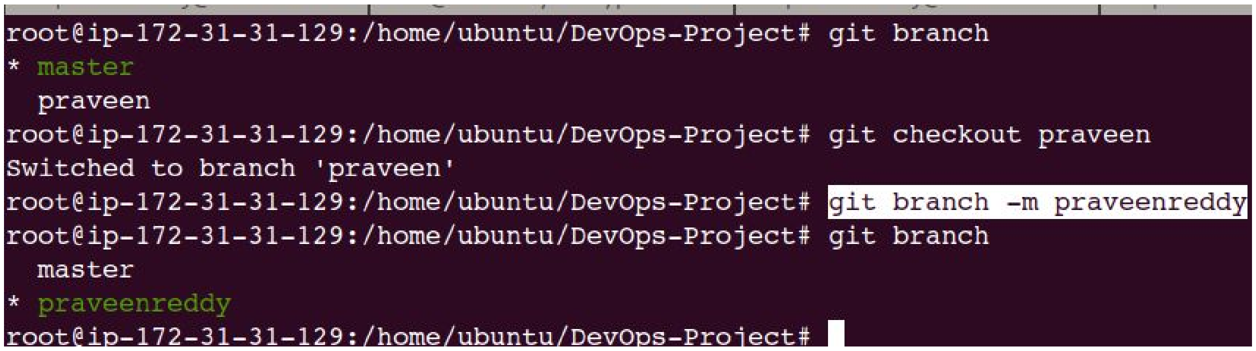
```
oot@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git checkout -b praveen master
witched to a new branch 'praveen'
oot@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git branch
 master
 praveen
oot@ip-172-31-31-129:/home/ubuntu/DevOps-Project# touch merge.txt
oot@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git add merge.txt
oot@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git commit -m "merging"
praveen d9f3a2c] merging
Committer: root <root@ip-172-31-31-129.us-west-2.compute.internal>
```



```
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git merge praveen
Updating b63f685..d9f3a2c
Fast-forward
 merge.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 merge.txt
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git push origin master
Username for 'https://github.com': jirrapraveenreddy
Password for 'https://jirrapraveenreddy@github.com':
Counting objects: 2, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 252 bytes | 252.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/jirrapraveenreddy/DevOps-Project.git
   b63f685..d9f3a2c  master -> master
```

Git Rename:-

> git branch -m praveenreddy



```
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git branch
* master
  praveen
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git checkout praveen
Switched to branch 'praveen'
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git branch -m praveenreddy
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git branch
  master
* praveenreddy
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project#
```

Git Move:-

> mkdir src

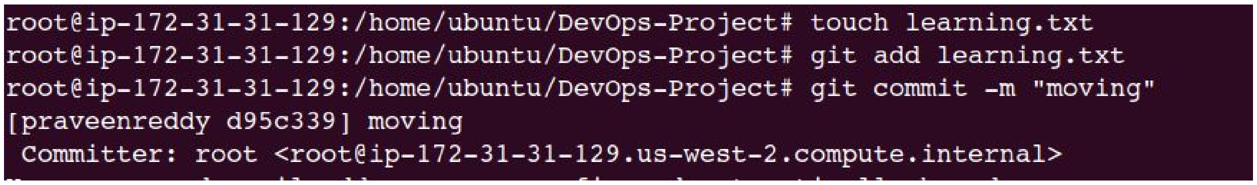
> touch learning.txt

> git add learning.txt

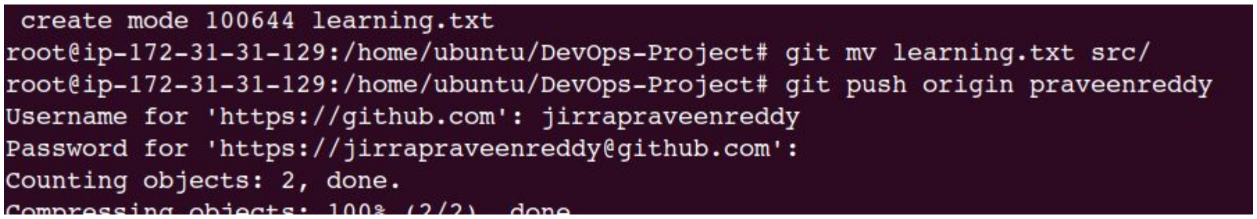
> git commit -m "moving"

> git mv learning.txt src/

> git push origin praveenreddy



```
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# touch learning.txt
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git add learning.txt
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git commit -m "moving"
[praveenreddy d95c339] moving
 Committer: root <root@ip-172-31-31-129.us-west-2.compute.internal>
```



```
 create mode 100644 learning.txt
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git mv learning.txt src/
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git push origin praveenreddy
Username for 'https://github.com': jirrapraveenreddy
Password for 'https://jirrapraveenreddy@github.com':
Counting objects: 2, done.
Compressing objects: 100% (2/2), done
```

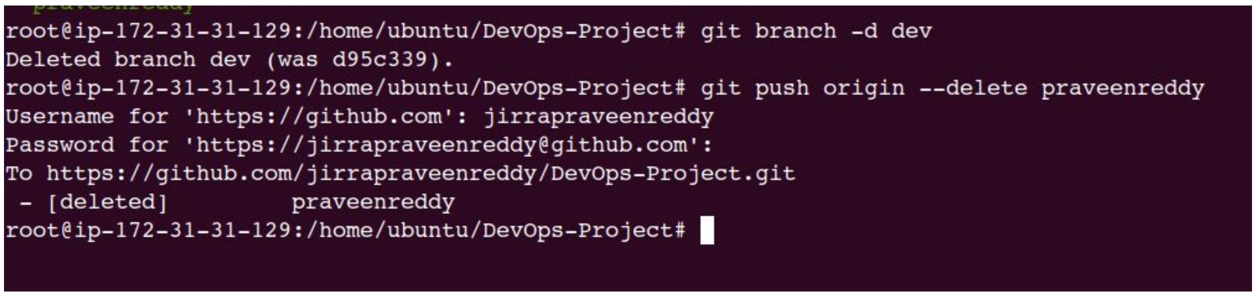
Remove the Branch:-

`// delete branch Locally`

`> git branch -d dev`

`// delete branch remotely`

`> git push origin --delete praveenreddy`



```
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git branch -d dev
Deleted branch dev (was d95c339).
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project# git push origin --delete praveenreddy
Username for 'https://github.com': jirrapraveenreddy
Password for 'https://jirrapraveenreddy@github.com':
To https://github.com/jirrapraveenreddy/DevOps-Project.git
 - [deleted]         praveenreddy
root@ip-172-31-31-129:/home/ubuntu/DevOps-Project#
```

Git Tag:-

```
 > git tag
```

// you can also search for tags that match a particular pattern

```
 > git tag -l "v1.8*"
 > git tag -a v1.4 -m "my version 1.4"
 > git tag
 > git show v1.4
```

// delete the tag

```
 > git tag -d v1.4
 > git tag
```

Git Log:-

```
> git log --all
> git log -3
> git log --author jirrapraveenreddy
> git log --after 1
> git log --before  0.days.ago
```

// To view a summary of the changes made in each commit

```
> git log --stat
```

// To just get the bare minimum information in a single line per commit, use the –oneline option.

```
> git log --oneline
```

Git Stash:-

when you’ve been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

As already mentioned, Git's Stash is meant as a temporary storage. When you're ready to continue where you left off, you can restore the saved state easily:

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time

```
> git status

> git stash

> git stash list

> git stash apply

> git stash pop
```