GIT

What is GIT ?

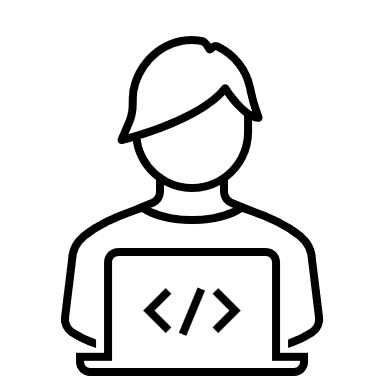
What is GITHUB?

Difference between GIT and GITHUB ?

**Before Version control system 🡺**

* No Audit tracking
* Difficult to rollback if anything goes wrong in prod.
* Not scalable for large teams.

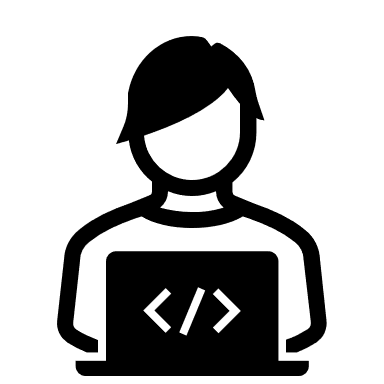
**Version control system 🡺**

 1.0

REPO

Application Server



 2.0

**Types of version control system :**

* Centralized
* Distributed

**Centralized version control system :**Diagram

Description automatically generated

**Distributed Version Control system 🡺**

Diagram

Description automatically generated

* Each developer has a local copy.
* More scalable than centralized version control system.
* Ex : Git , Mercurial

**Why GIT :**

* Open source. Its free and can be implemented in Kuberenetes, gitops and integration with Jenkins and other devops tools.
* Performant
* Detailed audit tracking

**Git VS GitHub :**

**Git :**

* Version control system
* Installed locally on the system
* Created in 2005 by Linus Torvalds
* Open source and used in multiple cloud repository services.

**Git hub :**

* Github is a GIT repository hosting services with other features.
* Runs on cloud.
* Created in 2008, currently owned by Microsoft.
* Not open source. Github is having both free and paid service.

**Ways to use GIT :**

* Command line interface.
* GUI
* Within devops tools.

**GIT Workflow:**

Diagram

Description automatically generated

**git diff**  🡺 Show difference between workspace and staging area

**git diff --staged**  🡺 difference between staging and local repo

**Branch and Merging :**

**Branch** :

* Branch can only be created from existing branch.
* Branches are references to commit.
* No code is copied
* Head determines on which branch you are on.

git switch feature1

git checkout feature1

A picture containing timeline

Description automatically generated

* To merge you need to switch to master branch.
* Then merge with feature 1 branch.
* If no changes made on master branch, it is very fast to merge . No conflict.

**git merge <name-of-the-branch-to-be-be-merged-into-the-branch-you-are-on>**

Diagram

Description automatically generated

* Here changes made on both the branches.
* Just moving the branch to look at new commit id won’t do merge.
* To merge in this scenario, we use recursive three-way merge.
* git switch master
* git merge feature1
* feature1 now merged with master. Git will choose fast forward or three-way merge automatically.

**git branch -d <branch-name>**

**git branch -d feature1**  🡺 it will delete the feature1 branch

Question :

Head 🡺 master

git switch feature1

git branch feature2

on which branch feature2 will be crated ?

Ans: feature1

git branch new\_branch old\_branch 🡺 new branch will be created under old branch.

**Git hub to Local :**

Graphical user interface

Description automatically generated

**git clone <remote-repo-link>**

* when you clone any remote repo to local repo, automatically branch main will be crated with commit id.
* Along with branch main, another branch is also created called origin/\* or origin/main
* If you copy all the branches from remote repo, origin will be crated for all the branches.
* Origin just tracks the remote branches.
* Origin/main
* Origin/feature1
* Origin/\* branch is for remote tracking only. Can not switch or commit in local.

**git branch -r**  🡺 it will display all origin remote branches.

**Git hub remote out of sync with local :**

* Once you clone the remote repo into your local. There may be a chance that some of your team member added or modified a file.
* In this case local repo is out of sync with remote .

Timeline

Description automatically generated

* Though new commit will be added into local with updated files from remote. The main branch will have the old commit.
* And the origin/main will be the new commit.
* We have to use merge in this case to merge both the branch.
* There way merging in case new files were created in both local and remote repo.

Timeline

Description automatically generated

**git branch - -merged**  🡺 show the branches merged with main branch.

**git branch --no-merged**  🡺 show the branches those are not merged.

**git branch -r**  🡺 display branches from remote repository

**git branch --all**  🡺 display branches from local and remote repository

**git remote -v**  🡺 display the remote or origin link which is mapped to local.

Merging and fetching from remote :

**git clone** [**https://your-repo-link**](https://your-repo-link) 🡺 it will create a folder inside your local repo with main branch and origin/main branch.

If you add any file in remote or create any branch in remote, you can do fetch.

**git fetch Origin**

🡺if you create file in local main and remote repo as well. You can do the below

git fetch origin

git merge origin/master

another scenario 🡺

* If you have another branch apart from main in remote.
* You can fetch remote to local
* Create a branch in local from the remote-another-branch
* Go to local master
* Merge with the branch you created in local.

git push origin head 🡺 Push the files to remote.

**Browsing history and commits:**

git log --all -oneline --stat 🡺 it will display the log and shows the modifications of file for each commit

git log --oneline --all --patch

git show <commit id>

git show <commitid : filename>

git ls -tree <commitid>

Graph tool :

* Graph tool is a third party tool.
* you can install it in visual studio code IDE by installing plugin.
* visual studio code > plugin > git graph
* once you install, in left side you would see source control
* in source control, click n repository and you would see view git graph\git graph would show the details history in graphical way.

**Remove files :**

**git ls-files**  🡺 shows the files in staging area

|  |  |  |
| --- | --- | --- |
| workspace | Staging | Local repository |
| a.txt  b.txt  ls -lrt  rm b.txt  by doing rm b.txt, it will delete from workspace, but file will be there in staging and local repo. In order to remove the file from staging you need to use add command and commit. | a.txt  b.txt  git ls-files  git add . | a.txt  b.txt  git commit -m “message” |

OR

**git rm file1**  🡺 it will remove from working area and staging as well.

+

**git commit -m “message”** 🡺 it will remove from local repo.

**Pull Request :**

* Pull request is the request to pull the changes from one branch to another branch.
* pull requests allows collaborators of the project to review, comment and update code.
* pull request is git hub specific term

A picture containing timeline

Description automatically generated

**Fork**

* Fork is used in GitHub. It means when you fork any repo, it would create a repo in you’re a/c.
* create a repo in your a/c
* once you modified the forked repo, you can do a pull request.

Timeline

Description automatically generated

**Keeping forked repo updated 🡺**

Fetch upstream 🡺

A picture containing timeline

Description automatically generated A picture containing Word

Description automatically generated

Graphical user interface, Word

Description automatically generated Graphical user interface

Description automatically generated with medium confidence

**Merge conflicts**

**merge conflict in GITHUB and GIT are diffrenet.**

**IN GITHUB 🡺**

Branch main 🡺 🡺 Branch -release-2.0

(modified file 1) (modified other content in file1)

in this case there would be merge conflict, if you want to merge.

* Git hub 🡺 go to your branch 🡺 pull request 🡺 It will throw an error “ cant automatically merge ‘
* if you scroll down, you would see error as this branch has conflict that must be resolved by webeditor or command line.
* open through web editor, remove the lines you don’t want.
* OK

**IN GIT 🡺**

mian branch : feature 1 branch

file1 file 1

content=xyz content = PQR

in this scenario, if you modify the same file in both main and other branch and you try to do merge. you would get conflict.

if you are in visual studio code, you can edit by editor and save by adding the new line in main-branch-file.

**Ignoring files :**

**.getignore file**

**.getignore file is a file that you xan define to ignore files.**

**workstation staging localrepo**

file1 file1 file1

file2 a.txt a.txt

log ADD 🡺 .gitignore .gitignore

a.txt

.gitignore

* If you define .gitignore and write file2 and log inside it, it will ignore both the file and not add or commit.
* but if file2 and log is already present in remote or local repo, GIT will not obey the .gitignore file.

**Markdown**

* Add formatting element to plain test document.
* default for showing readme in GitHub
* Basically, markdown document is a document if you write in by using certain syntaxes, the file will be formatted/styled in that way. It is like CSS stylesheet.

EX: newfile.md

#hello this is header

## 2nd header

GIT HUB Issues :

* github issues are like tasks/requests/bugs or any ticket kind of thing which is created by team lead.
* it’s a component of github.
* the issue assigned to the team member would work on that issue and close.

**GitHub webhook vs API:**

github 🡸 🡺 Jenkins 🡺 aws

git hub webhook is a post call to Jenkins , if there are any changes in repository.

Github Discard Integration :

Integration 🡺 webhook 🡺 bot

Github and Jenkins :

* job 🡺 configure 🡺 build trigger 🡺 github hook trigger for git scm polling
* github 🡺 repo 🡺 setting 🡺 webhook 🡺 add webhook

**Branching strategy :**

* trunk
* gitflow

**TRUNK**

* Trunk based development
* small frequent updates to a core trunk.
* code review, feedback done via pull request.
* feature branches are short lived.

**Gitflow** :

* specific roles to different branches.
* develop branch created from the main.
* release branch created from develop.
* feature 1 branch created from develop.
* when a feature is complete, that feature branch is merged with develop.
* release branch created from develop for final review.
* release branch merged to both main and develop.
* if an issue is found, hotfix created from main
* once fixed, hotfix merged to both develop and main

**Roll back - Git revert**

**can be done only in git CLI. Not in GUI.**

* every revert generates another commit id

git revert <commitid>

**Rollback changes with reset**

* it does not create additional commit id. It will just reset to previous commit id.

**git reset <commit2>** 🡺it will change the files from staging and local repo, but will keep the file in working directory. so you can use that file later if you want.

**git reset <commit id > --hard** 🡺 it will change files everywhere along with working directory.

**Rebase and Rebase vs merge**

* if we run a regular merge, it will create a new commit id. in this case commit history will be very messy.
* with rebase instead of creating a new commit id, it will just move the commit id of branch to main .
* creates cleaner commit history.
* no additional merge commits
* easier to navigate "git log"

disadvantages:

* rewrites commits
* rewrites history

**git switch feature1**

**git rebase main**  🡺 this will anchor all commits of feature 1 to main branch

**squash merge :**

* using squash merge , you can squash multiple commit into one commit

ex :

**git rebase -i HEAD~3** 🡺 top three commits will be squashed. It will open in vi editor. edit the file and replace pick with squash.

**Git cherry Pick :**

* if you want to add selected commit from any branch to main. you can use cherry pick

git cherry-pick <commitid>

git stash :

-it saves files in stash.

git stash save -m "comment " ==> it saves files of working directory in stash

git stash list ==> list of all files stored in stash along with stash id

git stash apply <stash id> ==> it restores the files from stash to master barnch of working directory. stash id can be 0,1,2...etc

**Installation**

**sudo apt-get git** 🡺 installing git in unix system

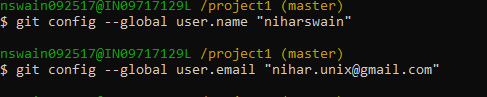
**git –version** 🡺 shows the version number of the git



**Configuring Git**

$ **git config --global user.name "King Kong"**

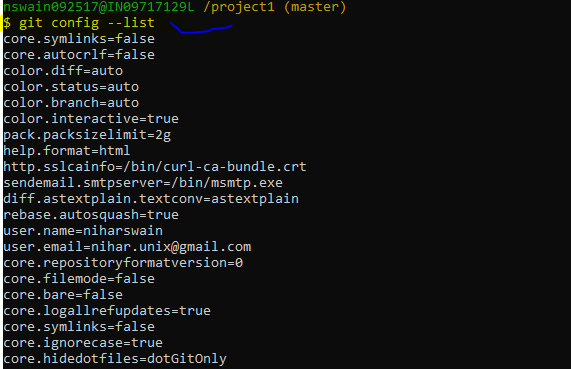
$ **git config --global user.email** [**king-kong@gmail.com**](mailto:king-kong@gmail.com)



**$ git config --list** 🡺 list out your git configuration details like your user id and mail id.

user.name=King Kong

[user.email=king-kong@gmail.com](mailto:user.email=king-kong@gmail.com)



**How GIT works 🡺**



**Creating GIT repository**

When we use GIT, we create one repository for each project that we work on. It is suggested to create all these project repositories under one directory called git.

Steps to create a repository for project 1

mkdir git

cd git

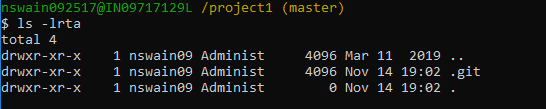
mkdir project1

cd project1

git init

**$ git init** 🡺 Initialized empty Git repository

* The "init" command stands for initialize. Once you run "git init",
* Git will initialize a hidden directory called ".git" in the project's root directory



* when git initializes it creates a hidden directory called .git in the root.
* You will see all sha1 files inside the object directory of .git folder.
* first two letters/numbers of the sha1 code is the file name in the object directory files.
* we can not read the file but we can use a low level pluming command to use it.
* .git >> objects >> multiple directory named as per the starting two letter of sha1 files >> go to any folder
* >> you will get file

drwxr-xr-x 1 nswain09 Administ 4096 May 14 21:39 4f

drwxr-xr-x 1 nswain09 Administ 8192 May 14 21:39 28

drwxr-xr-x 1 nswain09 Administ 4096 May 14 21:39 21

drwxr-xr-x 1 nswain09 Administ 8192 May 14 21:39 0e

drwxr-xr-x 1 nswain09 Administ 4096 May 14 21:39 05

drwxr-xr-x 1 nswain09 Administ 8192 May 14 21:55 13

drwxr-xr-x 1 nswain09 Administ 4096 May 14 21:56 cd

drwxr-xr-x 1 nswain09 Administ 4096 May 14 21:56 16

$ ls -lrt

total 17

-r--r--r-- 1 nswain09 Administ 689 May 14 21:39 14e82e869a0a94b4a34fa2f462c727066009ea

-r--r--r-- 1 nswain09 Administ 275 May 14 21:39 c444db7713fd74a0b4c5bcd78262fa972bebc1

-r--r--r-- 1 nswain09 Administ 13607 May 14 21:39 cb87a248de92d9b2b1ab692b592e772f0a98b2

-r--r--r-- 1 nswain09 Administ 6765 May 14 21:39 94741ea4413013f9cc9623eb81c0cc7493bc2f

-r--r--r-- 1 nswain09 Administ 250 May 14 21:39 037f912f940829951d900cd796c8720a997568

-r--r--r-- 1 nswain09 Administ 247 May 14 21:39 bb6ec493a2c0c233b6871197fb5600a0cfb8a0

-r--r--r-- 1 nswain09 Administ 540 May 14 21:39 faab66ed793f66bf64572c8238ab73e431f657

-r--r--r-- 1 nswain09 Administ 890 May 14 21:39 5d8c4296120ce85ba0a9591e93f14de895cb30

-r--r--r-- 1 nswain09 Administ 1506 May 14 21:39 20b1506517bc4d09f6bc4003c5eead56b44e88

-r--r--r-- 1 nswain09 Administ 522 May 14 21:39 da574883ffae4bd545fb2ec3adc03c75503b0b

-r--r--r-- 1 nswain09 Administ 144 May 14 21:39 224173070fcedf77ca4466b3ec0447e449c501

-r--r--r-- 1 nswain09 Administ 1384 May 14 21:39 a4638ff11c6745093b4f613dcfef6749022a7e

-r--r--r-- 1 nswain09 Administ 49 May 14 21:56 31f715b21d5bcada6f3d4f6c3b6eda474603c9

you can read the sha1 file using the below command.

**git cat-file 31f715b21d5bcada6f3d4f6c3b6eda474603c9**

$ **git status**

On branch master

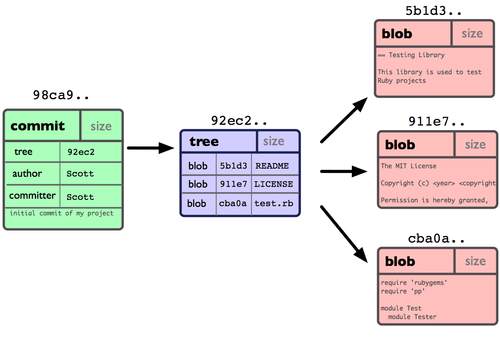
Initial commit

nothing to commit (create/copy files and use "git add" to track)

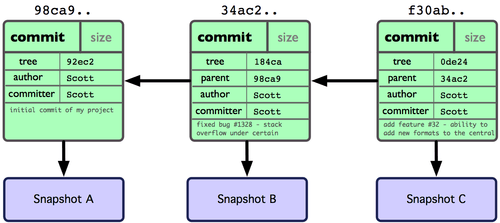
**Git Branches**

**How GIT stores its data ?**

* GIT stores data as a series of snapshots.
* When you commit, GIT stores a commit object that contains below
  + pointer to the snapshot
  + author
  + message meta data
  + zero or multiple pointers for commits or parent commits (zero parents for the first commit, 1 parent for the next commit, count changes as per the number of commits performed)
* Lets assume that we have a directory having 3 files. ( readme test.rb license)
* Once you add these in staging area and commit, it stores data as below
  + Git creates blob objects for each file present in the directory or files those were added in staging area.
  + GIT creates a blob called tree which contain the sha1 code and name of the files that we have committed.
  + GIT create a blob called commit which contains sha1 code of the correspondent tree, author name, committer name and the comment.

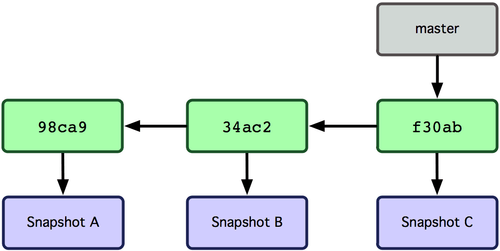


* If you make some changes and commit again. GIT will create blob files for each file, a new tree and a commit file pointing to the newly created tree.
* Below image shows the data if we create 2 commits after this.

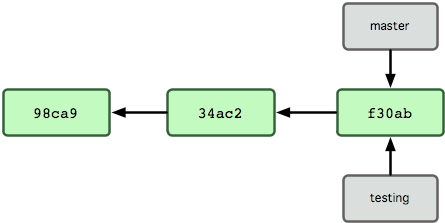


**What is branch?**

* A branch in GIT is a movable pointer to the snapshot or one of the commits that we did.
* The default branch in GIT is master.
* So Master branch points to the last committed snapshot.
* Every time we commit, the master branch moves its pointer automatically to the latest committed snapshot.

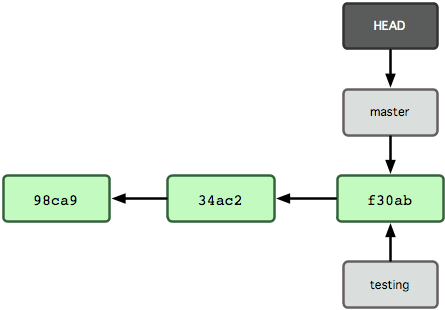


* Lets create a branch called testing. Syntax : **git branch testing**
* Creating a branch will create a pointer to the latest snapshot.

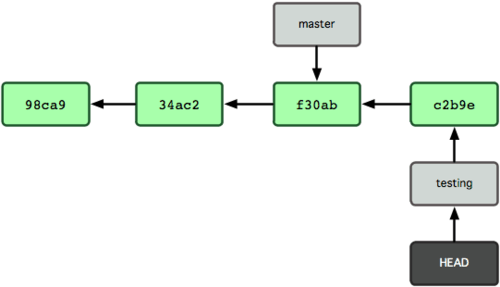


**What is HEAD?**

* HEAD is a special pointer which points to the current working branch.
* By default, head points to master. So, in the above example even after we have created a new branch testing, the head branch is master.



* We must use checkout command to move the head pointer to any specific branch.



Branch commands

**$ git branch testing**  🡺 Create a new branch named testing

**$ git checkout testing**  🡺 moves head to testing branch.

**Add files to staging area**

git add 1.txt 🡺 adding a single file

git add 1.txt 2.txt 3.txt 🡺 adding multiple files in staging area.

git add . 🡺 add all files in the directory

git add --all 🡺 option "--all" tells Git: "Find all new and updated files everywhere throughout the project

and add them to the staging area

git add -A 🡺 option "--all" tells Git: "Find all new and updated files everywhere throughout the project

and add them to the staging area

**Remove files from staging area**

**git rm --cached 1.txt** 🡺 remove the file from staging area

**git reset 2.txt** 🡺 reset is opposite to add. it will remove the file from staging area.

**git reset** 🡺 it will remove all the files.

$ **git reset --soft HEAD^** 🡺 The "--soft" option means that the commit is cancelled and moved before HEAD

$ **git add file-i-forgot-to-add.html**

$ **git commit --amend -m "Add the remaining file"**

To understand what that "HEAD" thing represents, recall that we work in branches.

Currently we're in the master branch, and HEAD points to this master branch. When we switch to a different branch later,

HEAD will point to that different branch. HEAD is just a pointer to a branch:

**committing files in GIT**

$ git commit -m "Add three files" ==> the "-m" option, which stands for "message"

[master (root-commit) abfbdeb] Add three files

3 files changed, 0 insertions(+), 0 deletions(-)

create mode 100644 another\_file.js

create mode 100644 my\_new\_file.txt

create mode 100644 new\_file.rb

The message tells us that there have been three files added to the current branch,

which in our example is the master or the main branch

The "create mode 100644" message tells us that these files are regular non-executable files.

The "0 insertions(+)" and "0 deletions(-)" messages mean we haven't added any new code or removed any code

from the files

**$ git commit -a -m "Do something once more" 🡺add and commit files simultaneously**

what if you forget a file to commit. Below is the process

$ git reset --soft HEAD^

$ git reset --soft HEAD^ ==> The "--soft" option means that the commit is canceled and moved before HEAD

$ git add file-i-forgot-to-add.html

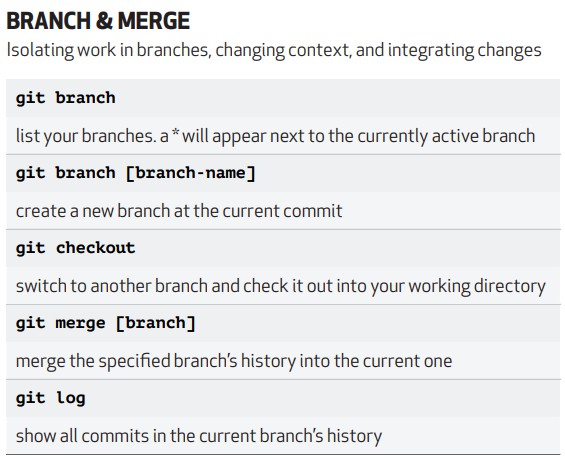
$ git commit --amend -m "Add the remaining file"

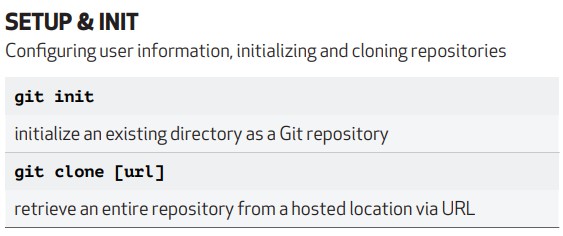
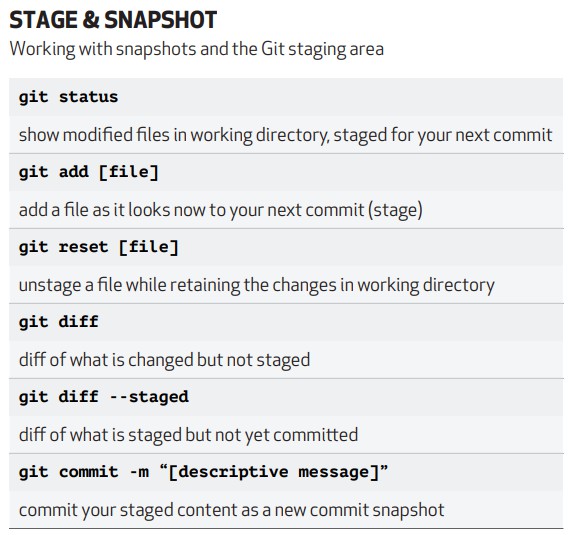
The "--amend" option lets you amend the last commit by adding a new file (or multiple files).

Using the "--amend" option, you can also overwrite the message of your last commit

Table

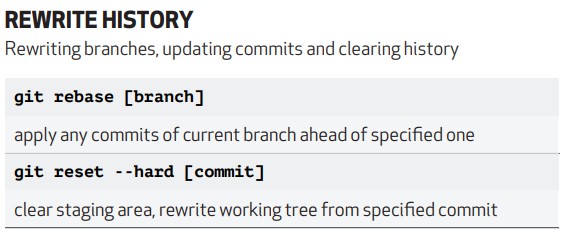
Description automatically generated

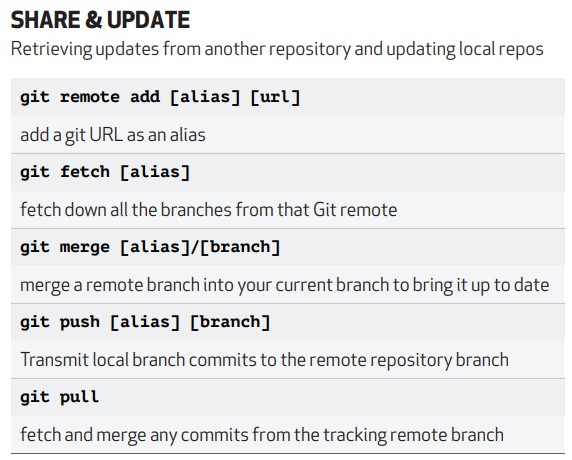




Table

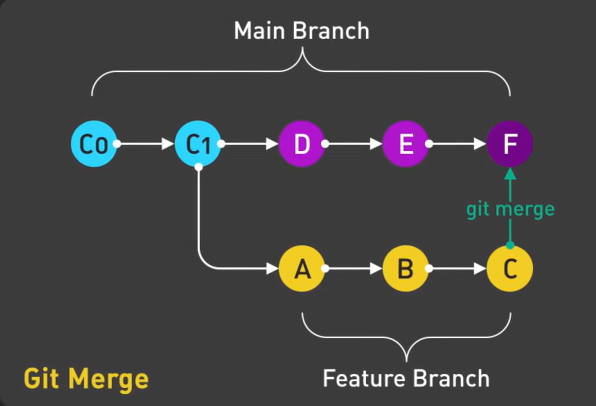
Description automatically generated

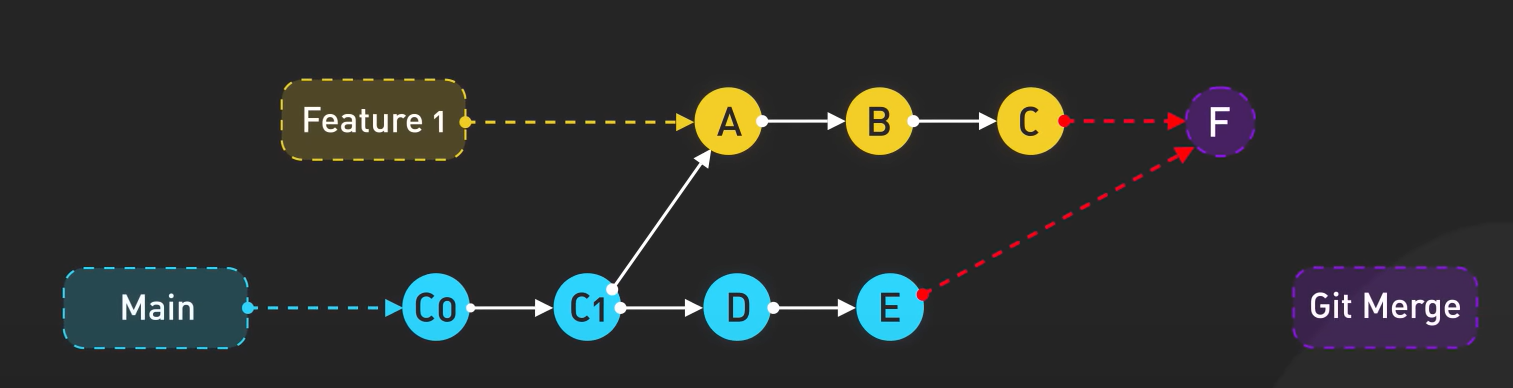


Graphical user interface, text, application

Description automatically generated

Difference between Rebase and merge ?

* + Git merge merges the latest commit of feature branch to main branch
  + Gt rebase merges all commits to main branch
  + 



A diagram of a diagram

Description automatically generated

A diagram of a branch

Description automatically generated

Squash

A diagram of a flowchart

Description automatically generated

A diagram of a branch

Description automatically generated