git is a distributed version control system and not a central version control system

git --version # to check the version of the git

configuring git

we can define the settings in three levels

1. system level All users

2. global level All repositories of the current user

3. local level the current repository

git config --global user.name "aditya shashanka"

git config --global user.email adityashashanka@gmail.com #here we don't have a spance in email so we don't need double quotes

by default you will get vim editor but you can config your editor if you don't loke vim

git config --global core.editor "code --wait" # only works when the vs code pops up when you type code and enter in you terminal

git config --global -e # to open global settings in default editor

now let's configure how git should handle end of lines. On windows end of lines are marked with two special charectrs chariage return(\r) and line feed(/n) on mac and linux end of lines are indicated with line feed i.e /n to prevent this we have to configure a property called core.autocrlf which will atomatically add and remove them depending on os

git config --global core.autocrlf input #shoulb be set to true if it's windows and input if it's mac/linux

git config --help # to get the info about the config command

git config -h # gives us a short symmery of the command and it's options

================================================================================================================================

2.0

let's create a directory for our project

let's create a directory called moon and go into it

we need to initiate a new repository to do that we type

git init

git workflow:

When we are ready to upload the changes we put it into staging area/index and review our changes then we will make a commit

eg: lets create 3 files named file1, file2, file3

we use add command to add them to staging area and review it and then everything is good so we commit it with the command

git commit -m "initial commit"

after you commit the staging area doen't become empty it will still contain the previous files

if you have two files f1 and f2 if you delete the single file f2 and make changes with the file f1 and add only f1 to the staging area and commit it the new f1 file and the old f2 file gets into the repository. to delete the file f2 from the staging area we need to add the file2 to the staging area and when there is no file named f2 it will reailise that git needs to delete the file from the staging area

git status # to see the status

git add file\_name.txt

git add . # to add all the files in the folder to the repository

git commit -am "commiting without staging"

git commit -a -m "commiting without staging in another way"

git ls-fiels #to see the files in our staging area

to remove a file from working directory and staging area we need to type

git rm \*.txt

.gitignore is used for those files for git to ignore

you need to list all the file names that you dont want to be in the repository

git status -s # to see the short status

git diff --staged #shows the difference between files from previous staging area files to present

git diff #shows the difference between the code on staging area and directory

git config --global diff.tool vscode

git config --global diff.tool vscode

git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOTE"

#when we use wait it will wait until we close the vscode $LOCAL $ REMOTE are place holders

git config --global -e

git difftool

git difftrool --stage

git log #to view our history of commits

git log --oneline #shows short summery of the commit

git log --oneline --reverse #to reverse the order of the logs

to see the content of the commmit we use show command

git show COMMIT\_UNIQ\_IDENTIFIER

insted of commit unique identifier we can also use the head pointer

git show HEAD

git show HEAD~1 # to show one step back commit of head

to see exact version stored in a perticular commit without seeing diff we use :.gitignore

to see all the files and directories in a commit

git ls-tree HEAD

files are represented using blobs and directories are represented using trees

we can also use git show to see a perticular file in a commit with it's id

git objects:

commits

blobs

trees

tags

git restore --staged file1.js

to restore the files from the previous staging area

git restore --staged .

Discard the staging area

git restore file1

git restore .

# restore all files from staging area

git clean

# to remove all the untracked files from the working directory

git rm file\_name

#to remove the file from working directory and staging area

after removing the file from staging area and working directoy let's say we have committed it

to resore the file lets do this

lets see the logs

git log --oneline

git restore --source=HEAD~1 file\_name

now see git status -s to see the untracked files which means it directly restores the file on to current working directory but not to the staging area.

================================================================================================================================

3.

git log to view the commits

git log --oneline

git log --oneline --stat

#to view all the files have been changed

git log --stat

to see the difference we use git log --oneline --patch

in real time we have thousands of files and it's tough to see the commits with git log. we can filter commits on different things

git log --oneline -3

#to see the last three commits

git log --oneline --author="adityashashanka"

git log --oneline --after="2022-11-23"

git log --oneline --before="2022-22-23"

git log --oneline --after="yesterday"

git log --oneline --after="one week ago"

git log --oneline --grep="GUI"

#filters all the commits with the word GUI in the commit message

git log --oneline -S"OBJECTIVES"

#this command will give the results of all the commits that have word OBJECTIVES in it

to have the commits only from the perticular range we use command

git log --oneline fb0d184..edb3984

fb0d184 and edb3984 are the commit ids from and to commit ids

if you want to find all the commits that modTified perticular file we use the following commands

git log --oneline filename.txt

if the git finds the file name ambigeous the use the following command

git log --oneline -- filename.txt

format the output

git log --pretty=format:"hello%an" #an means author name. this command prints out hello aditya for all the commits

git log --pretty=format:"%an commited %H"

git log --pretty=format:"%an commited %h"

to see a perticular commit

git show commitid

git show HEAD

git show HEAD~1

to see the file version that is stored in perticular commit lets say in two commits before HEAD i.e HEAD~2

git show HEAD~2:filename.txt

git show HEAD~2 --name-status

#

to see the changes across two commits do this

git diff HEAD HEAD~2

git diff HEAD HEAD~2 filename.txt

git diff HEAD HEAD~2 --name-only

#to see the list of file names that are changed.

TO RESTORE OUR WORKING DIRECTORY TO A PERTICULAR COMMIT

git checkout commit\_id

head and master points to the latest commit we create

when we checkout a perticular commit the head will move to that perticular commit. this is what we call detached head

head is not pointing to a latest commit anymore when we checkout

after we checkout we shouldn't create a new commit. we should only view our code

if we create a commit at somepoint we should get back the head pointer to the branch and then the commit becomes non reachable(check the diagram 1.2 from this folder)

we did git checkout so now head is in the commit stage

if we not type git log --oneline we will only see the commits we did before that commit and we wont see the master

to view all we type

git log --oneline --all

to get the HEAD back to the MASTER we type

git checkout MASTER

finding bugs using bisect

git bisect start

to find the bad commit

git bisect bad

git shortlog #to see all the people contributed to our project

to restore a file from a perticular commit

git checkout a643k46 toc.txt

blame tool to find the author of the perticular file

git blame audiance.txt

tags:

lets say the last commit represents first version of our software so we can give it a tag or custom label

git tag v1.0 commit\_id

we can also checkout a commit using its tag

git checkout v1.0

to see all the tags we type git tag

to create an annotated tag we must do the following

git tag -a v1.1 -m "my msg 1.1"

with annotated tag we can assosiate a msg with that tag

if we now type git tag we won't see any msg with that tag 1.1

to see that msg we must type the command git tag -n

git show 2.2

git tag -d tag\_name # to delete a tag

skipped 3.16 and 3.17

================================================================================================================================

4.Branching

git works differently from subversion. when we create a new branch in subversion it copies all the files from the mater which makes it slow

git works differently it works with pointers master is a pointer to the commit

if we create a new branch new pointer is created at master snapshot

git knows which pointer we are working on with HEAD

Lets create a branch called bugfix

To see the list of branches we type git branch

Git branch bugfix

#above command creates a branch called bugfix

git branch #this command gives all the branches

git status #to see the branch we are on

Git switch bugfix # to switch to the bugfix branch

#to rename the branch we use

Git branch -m bugfix aditya-testbranch

#the above branch will change the name from the bugfix to aditya-testbranch

Git switch master #to switch to the master brancch

Git log --oneline --all

To delete the branch

Git branch -d aditya-testbranch

To delete the branch which isn’t merged we need to do force delte

Git branch -D aditya-testbranch

To see all the branches that are there in aditya-testbranch and not in master

Git log master..aditya-testbranch

If you are in master

Git diff aditya-testbranch

Git diff --name-only aditya-testbranch

Shows the names of the files that are there in aditya-testbranch and not in master

Git stash:

When you make some changes to the files in working directory and didn’t commit and trying to switcch b/n branches it wont allow you to do so. If you want to switch without commiting you need to stash them

Git statsh push -m “New tax rules”

By default new untracked files are not included in the stash

Git stash list

Git stash show stash@{0}

Git stash show 1

git stash apply 0

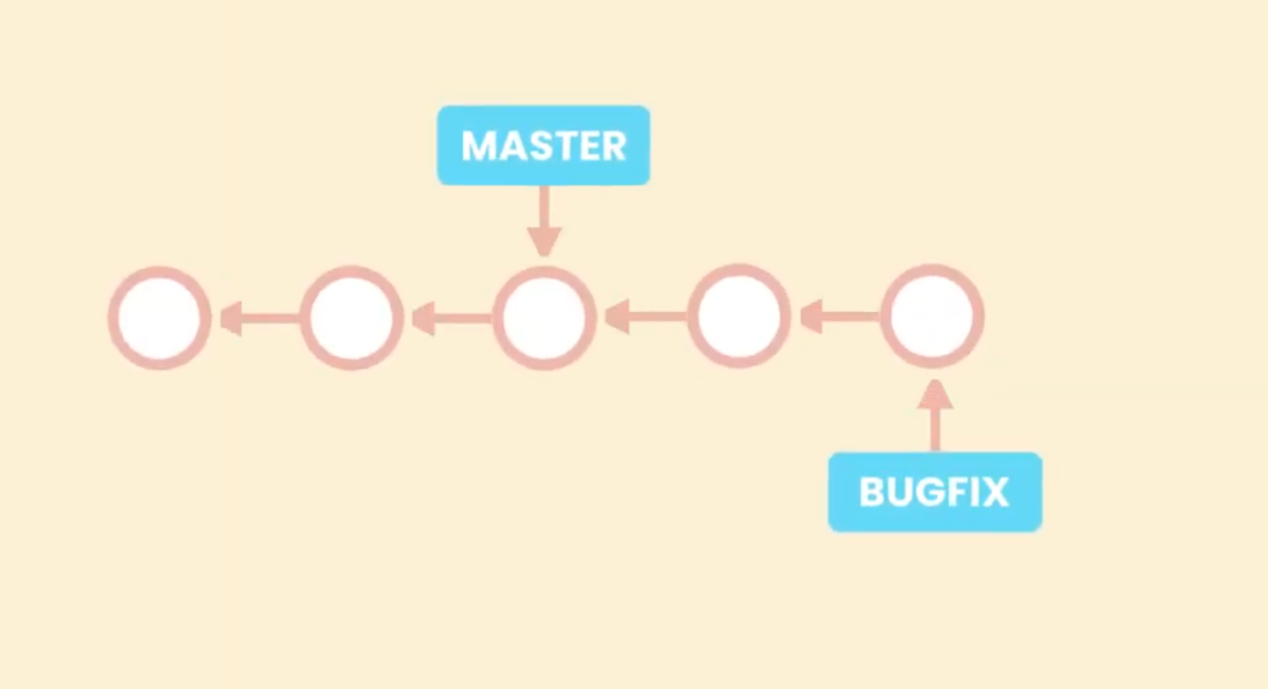
Git stash drop 0

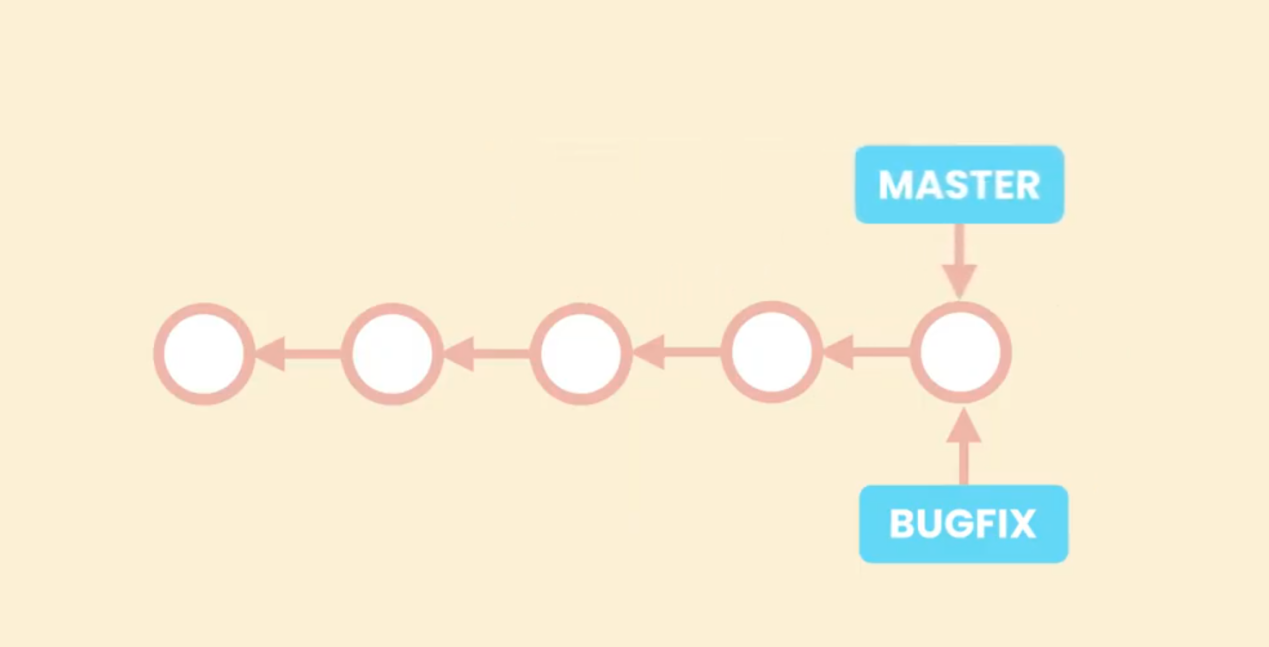
Merges:

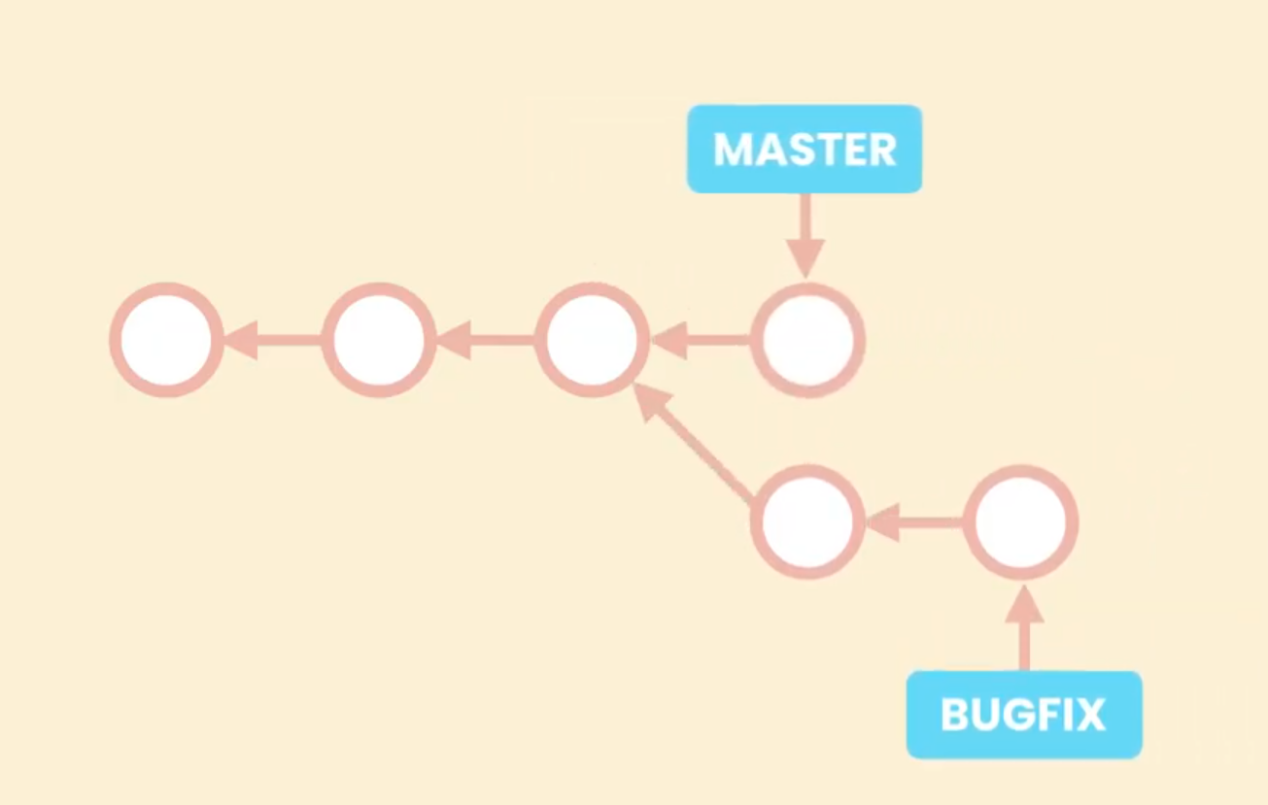
There are two types of merges

1. Fast forward merges
2. 3-way merges
3. Fast forward merge:

When we have commits in linear path we use this type of merge. The pointer moves direcly to the present

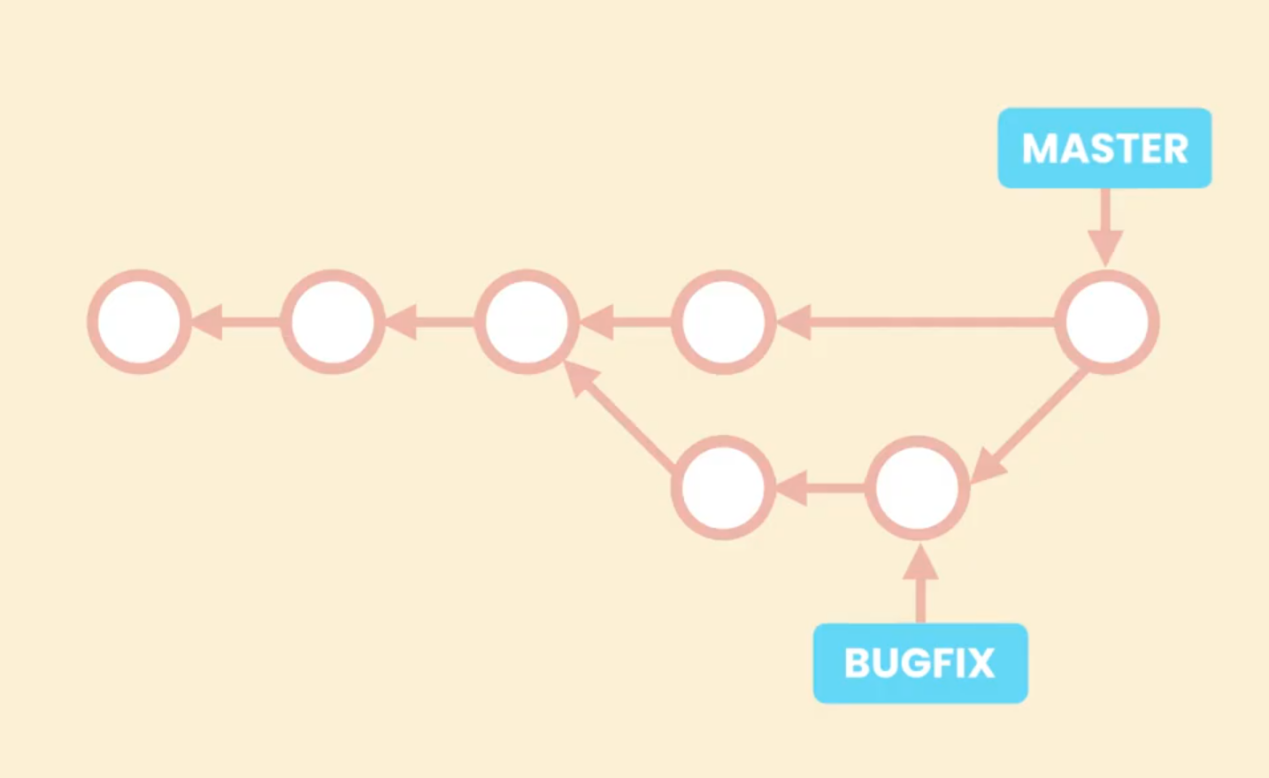




1. Three way merge:

In this type the branch is non -linear I.e after the test branch there is another commit

In this case if the pointer of the master comes to bugfix then the changes done on the next commit of the master branch is lost. So git create new commit that combines the changes of the both.



It is called three way merge because it looks at three differeent snapshots

The point at which the branch is divided, the point of the latest master branch and latest new branch.

**Git log --oneline --all --graph**

Will give us the graph output of the branches. If you don’t mention all option it will one show the logs of the current branch

For linear path:

Go to master

Git merge test-branch

Lets commit the linear commit using no fast forward option

To create and switch to a new branch

Git switch -c new-test

Modify a file

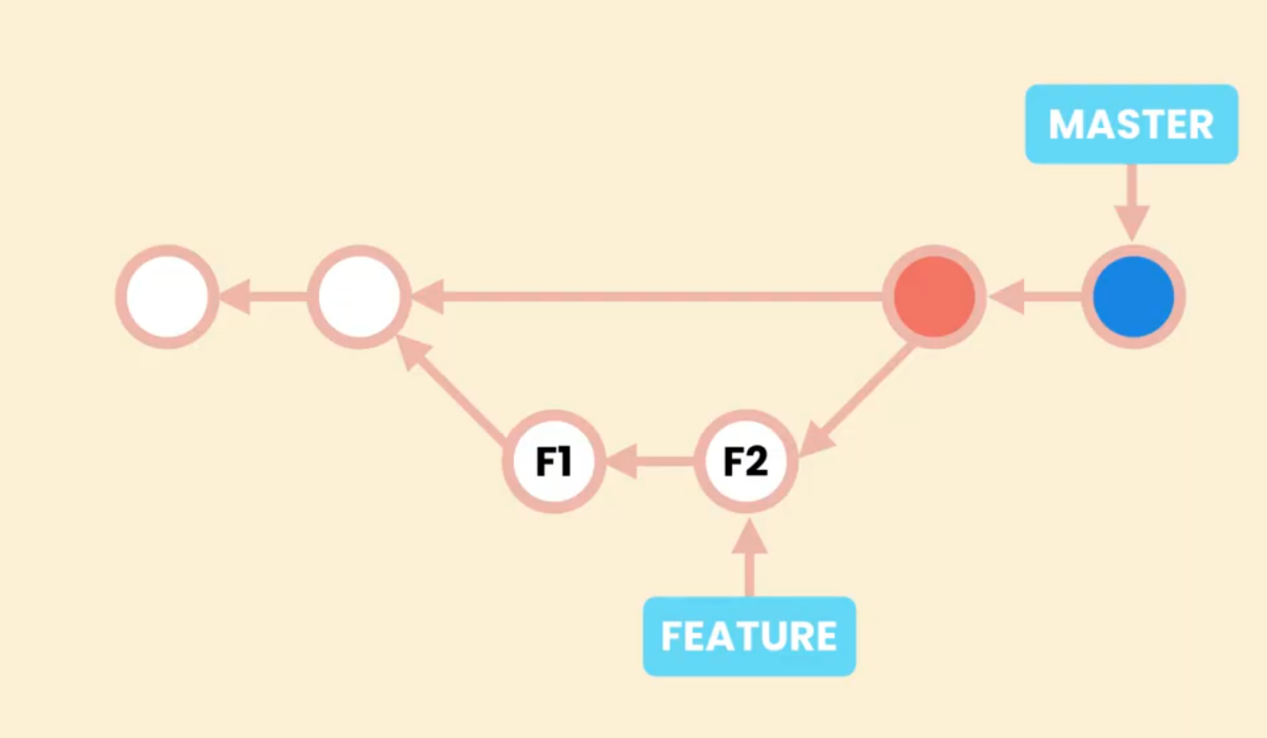
Commit it

Lets switch to master

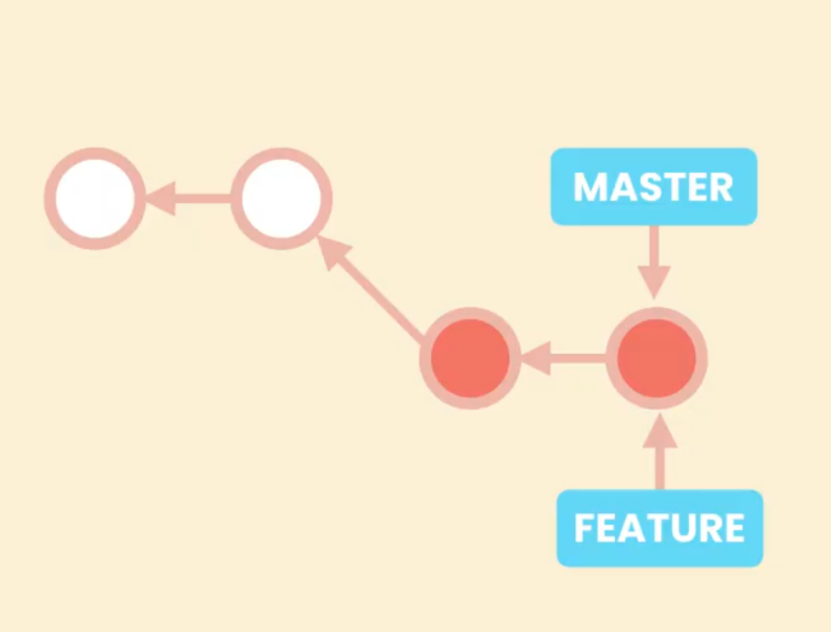
Git merge --no-ff new-test

Git log --oneline --all --graph

If you have used fasforward it gets tough I.e you have to goo many stages backwards to revert back the commit if the feature in previous branch fails. But if you use no fastfarword then if the feature fails it is easy to revert back to that commit because it’s just one step



NO FASTFARWORD(--no-ff)

Fastforward

We can disable fastforward merges for all our repositories or for perticular repositories.

##### Git config --global ff no

$ git config --global --add merge.ff false

1. Way merges:

Let’s create a new branch and switch into that

Git switch -c adiie-test

Edit a file and commit it

Switch to main and edit the file and commit it. Then do the merging

Git branch -d branch\_name # to delete the branch

Git branch --no-merged #shows the branches that aren’t merged

Merging conflicts:

Conflicts happen when

- the same code is changed in different ways in two branches

- if you given a change in one branch and deleted in another

- when the same file is added in two different branches but the content of the file is different

Lets create a branch and move into It and edit a file

And switch to master again and edit it

Try to do git merge now you see the confilct

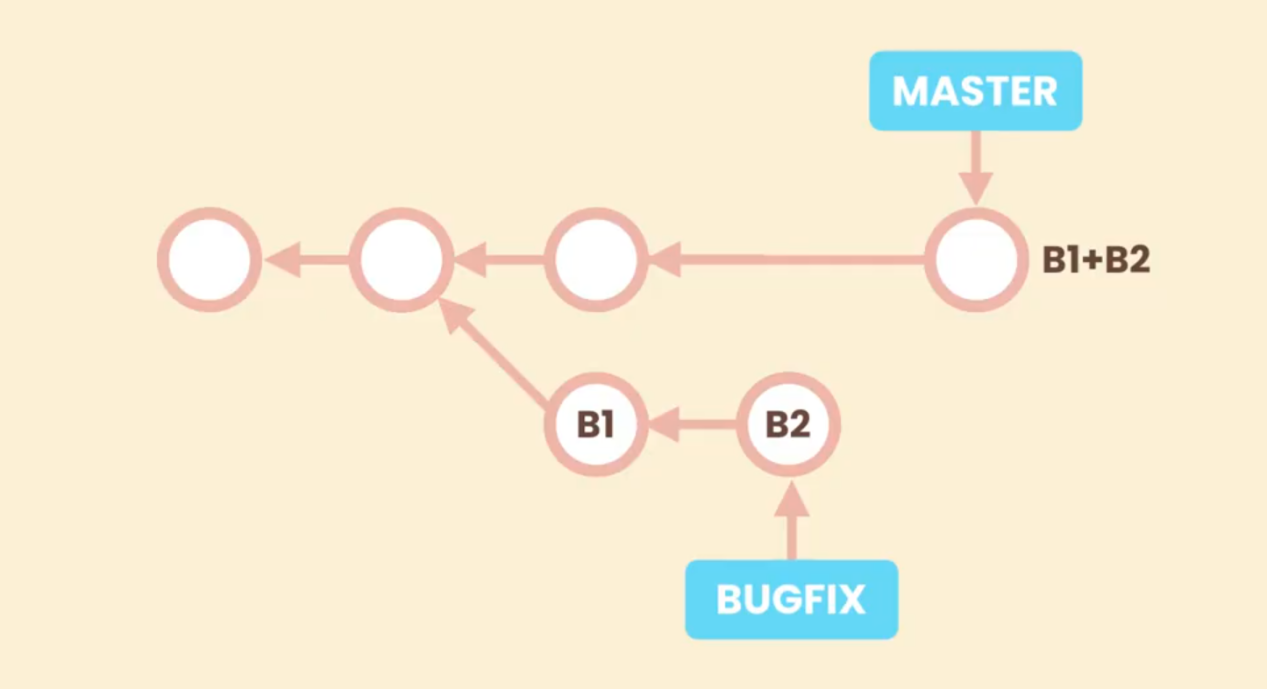
Do git status

Now resolve the confict by opening the file. You must remember that you shoudn’t add anything at this time

Now add the file to the staging area and commit it

**Squash merging:**

when you don’t want commits from a new bracnch but wants results from it we use squash merging



B1 ans b2 commits doesn’t look neat so we don’t want them in the history so we do squash merging which will give us result of the merge of previous mmaster and b1 and b2

Git merge --squash branch\_name\_of\_b1andb2

Now the files are modifieed and staged intomaster now we have to commit

After you do this remove the branch bcoz if you type git branch --no-merge it still shows this branch

You might think later that this branch isn’’t merged and remerge it

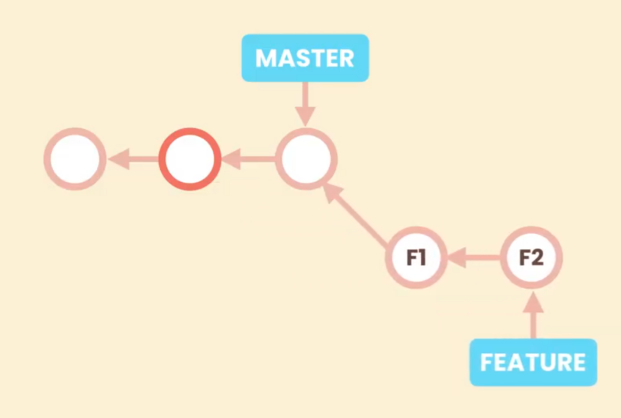
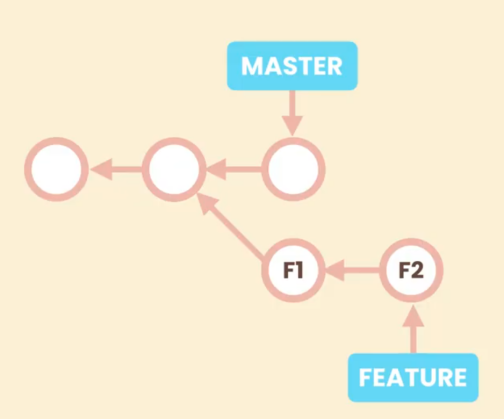
Git branch -d branch\_name

To force it use D

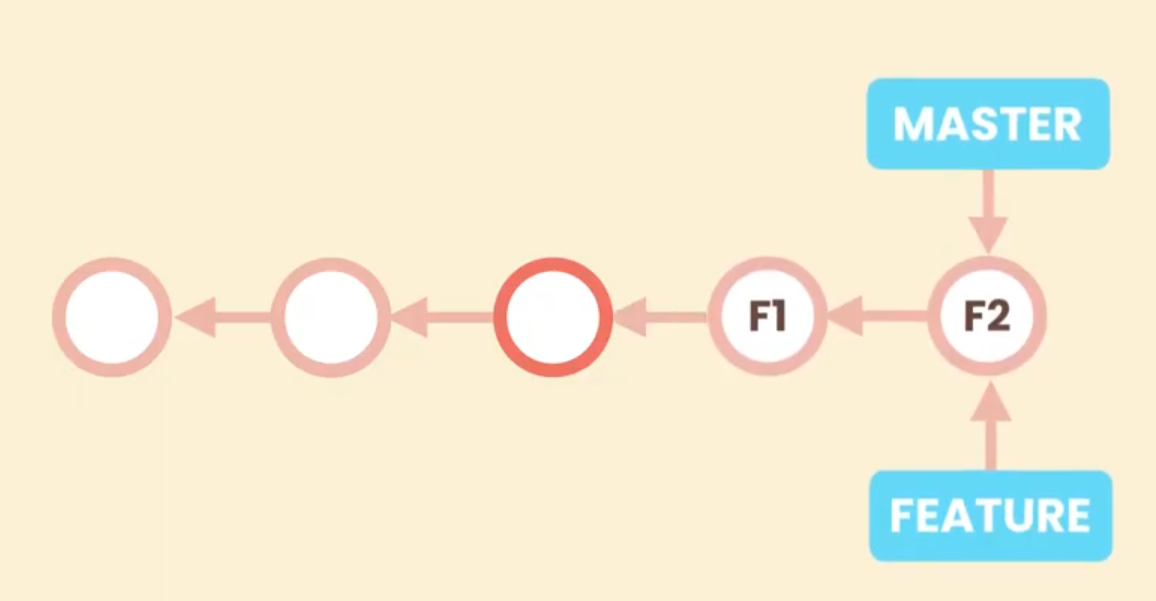
Git branch -D branch\_name

Rebasing:

Doing --no-ff will result in dirty history to get rid of it we use rebasih



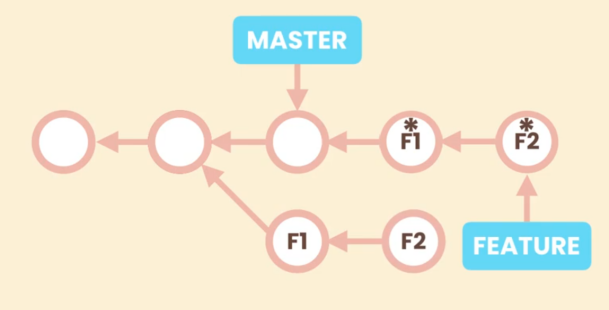
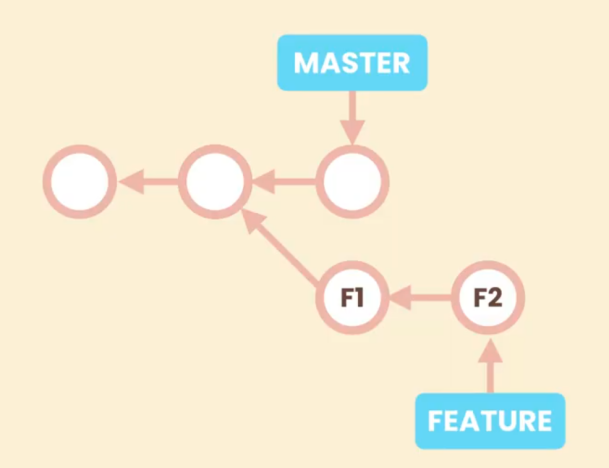
Now to do branch merging we can do fastforward merging



Whilie this sounds like a great idea you need to be cautious with rebasing because rebasing rewrites history. You should use rebasing only for the branches that are local in your repository. If you have shared this repositoy with your team if you have pushed or changes you shouldn’t use rebasing

When you do the rebasing the git isn’t going to change the base of f1 commits in git are immutable

So basically what git is doing is it’s going to create new commits that look like f1 and f2 and then it’s going to move the feature pointer. Git rewrites the history



**Squash is used when we don’t need history of those commits where as rebase is used when you need history and you want it to be clean normal merge is used if you want history and fast but it’s ok to be dirty**

To do the rebase switcch to the feature branch and do the following

Git rebase master

If you have a conflict and if you resolve the confict then you need to type git rebase --continue

Git rebase --skip

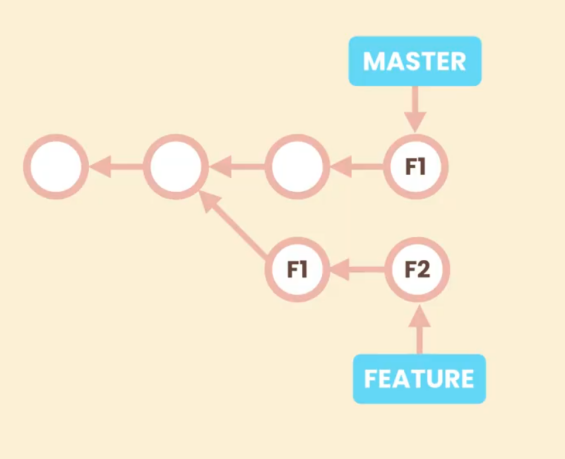
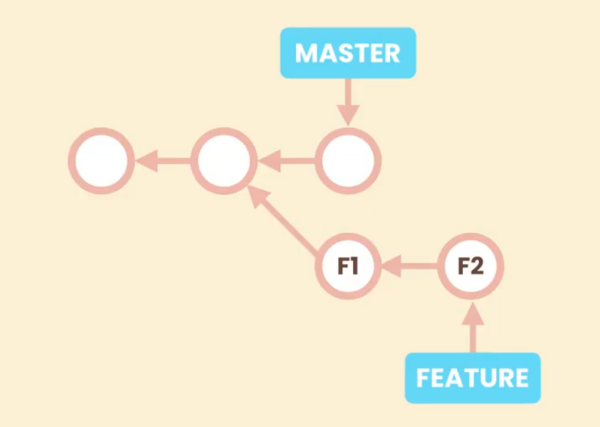
#will skip the current commit

Git rebase --abort

#if you don’t have a time for resolving conflict and end the rebasing you can do this now

**Cherry picking:**

Let’s say we have a commits f1 and f2 in feature branch. We want f1 commit to be merged to master but not f2 since the f2 is not ready so we use cherry picking.



Git cherry-pick commit-id

#git mergetool is used to conflicts .

How to retrieve a file a commit of a branch:

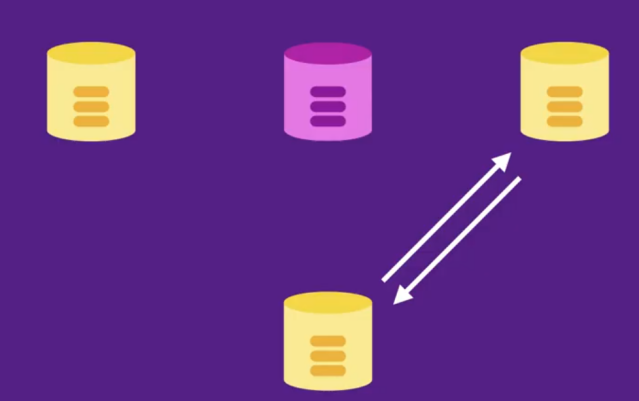
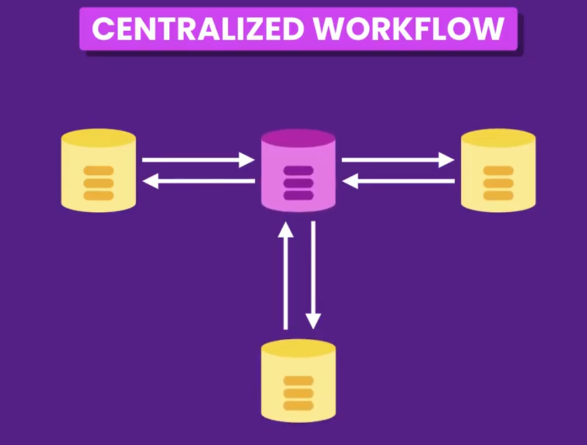
Git restore --source=feature/send-email file2

#feature/send-email is the branch name file2 is the file name

If git can’t interpret the file name add -- infront of the file name

## WORKLOADS:

Git is a distributed version control system with centralized workflow



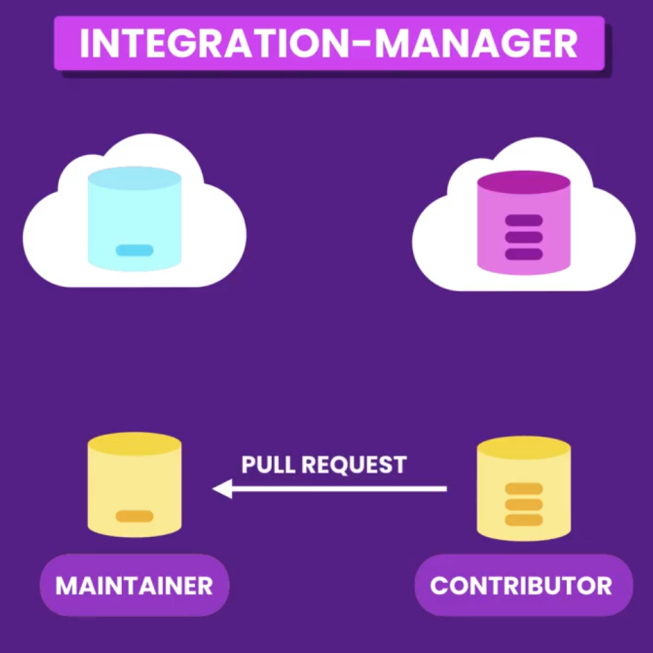
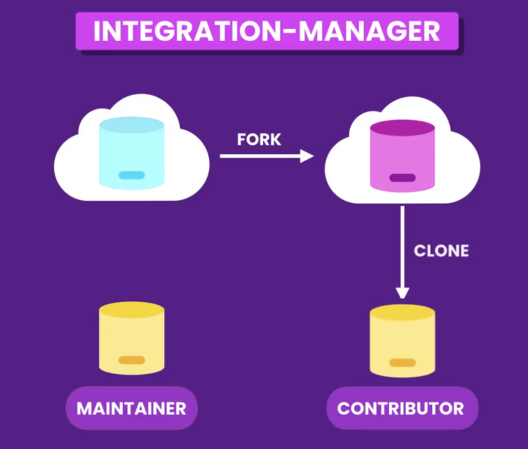
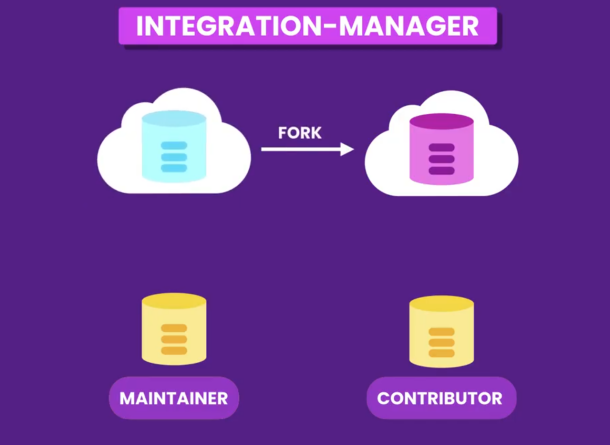
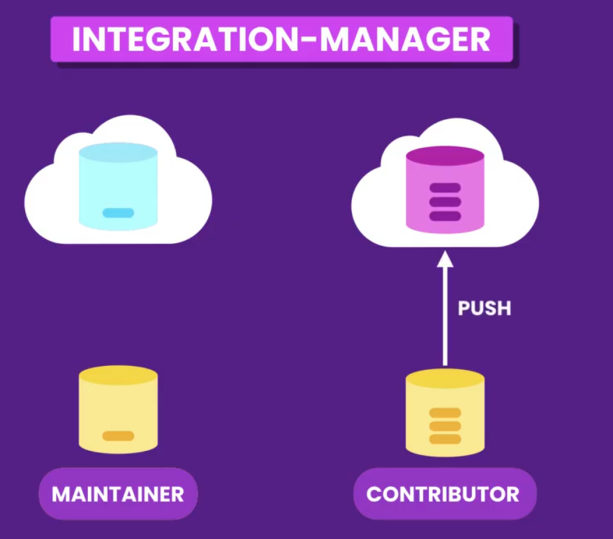
This is differnet from centralised version control system. Even if the main server GOES down we can work with our local repository and syncronize them

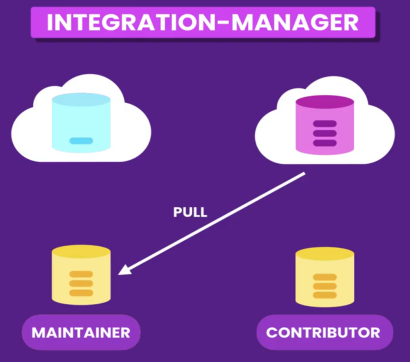
We can use our private local server as central repository or cloud repositories like github,gitlab,bitbucket

There is another workflow called integration manager workflow:

In an open source project we have one or more maintainers and many contributors. In open source project we have one or more maintainers and many contributers. THe problem is we don’t know these contributers

So to contribute we should fork the repository from the cloud next we clone this repository to copy on our mechine. We make a few commits and push to our fork repository and then send a pull request for the maintainer of the project. Maintainer gets notified and then they can pull in our changes review them and they like it they will update it into official repository





Create a git hub account and add a new repository

Copy the code with https

Git clone url\_copied

If you go inside the directory and type git log --oneline --all --graph

You will see head -> master, origin/master, origin/HEAD

When we clone a repository git names the repository as Origin. Origin is a reference to that repository

Master tells us where is the master branch in that repository

We can’t check into it . git switch origin/master

Git remote #this command shows this shows the list of remote repositories

#remote repositories are those which doesn’t stay in our local repositories

If we type

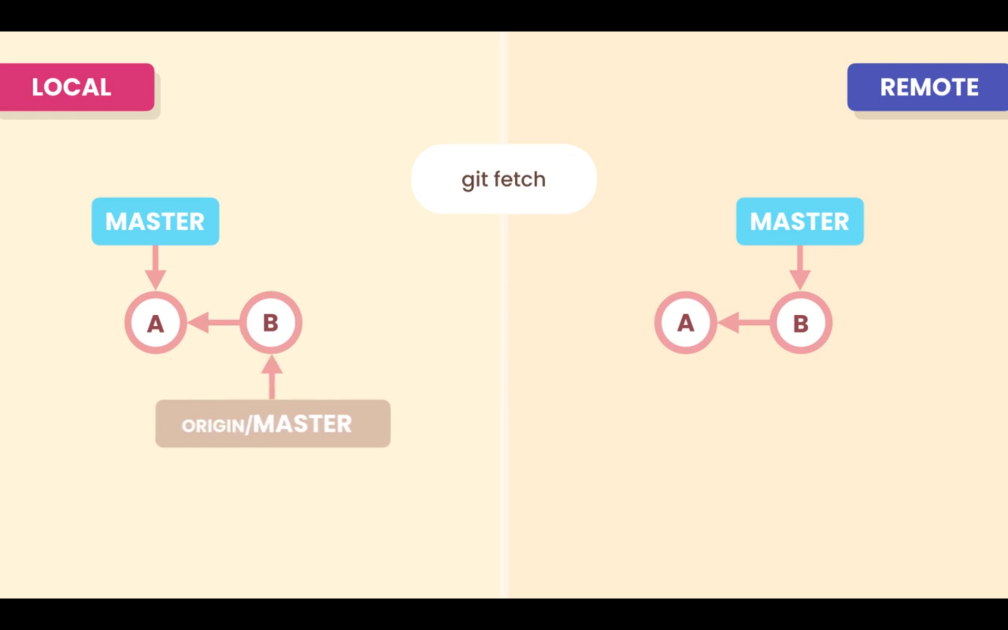
Git remote -v # we get more verbose output about the git repository

One thing we need to understand our local repository is not connected to remote repository

When we make a new commit in our remote repository our local repository will not be aware of that

To get this to our local repository we gotta use

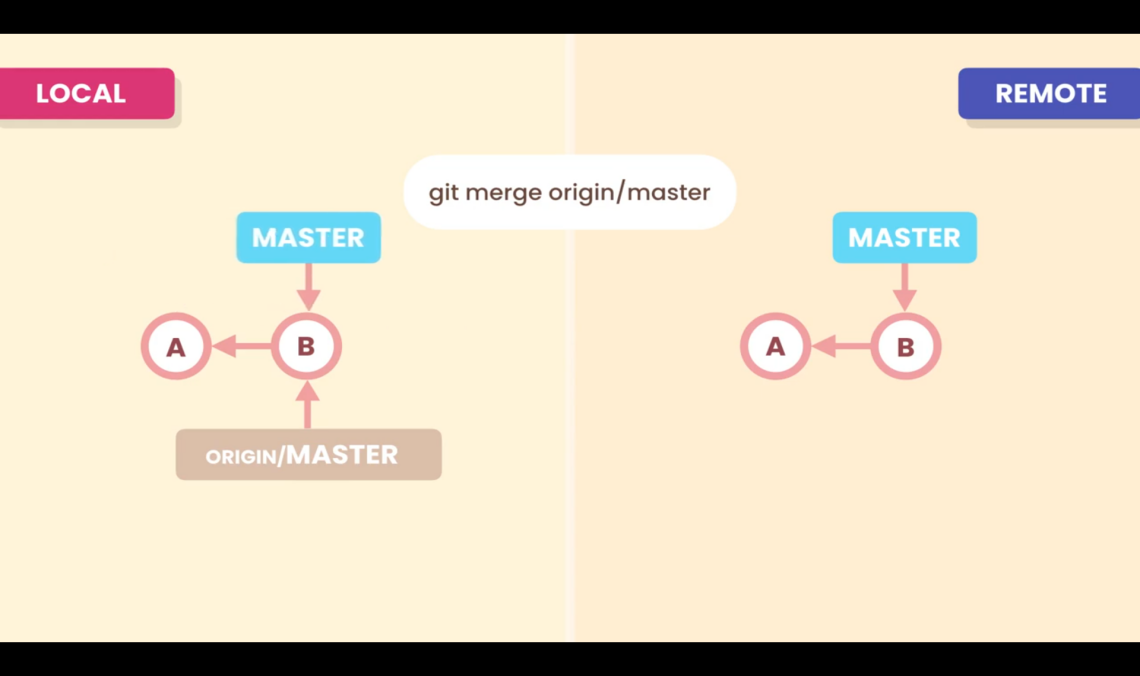
Git fetch



Even though we did the git fetch our local repo’s master is not updated only origin/master is moved

We go to the master and typee

Git merge origin/master



Git branch -vv

Git fetch #to fetch origin repos files

Git merge origin/master

Pulling:

When we want to do fetch+merge we gotta use pull

git pull

Git reset --hard HEAD~1

#will resed to previous commit

PUshing:

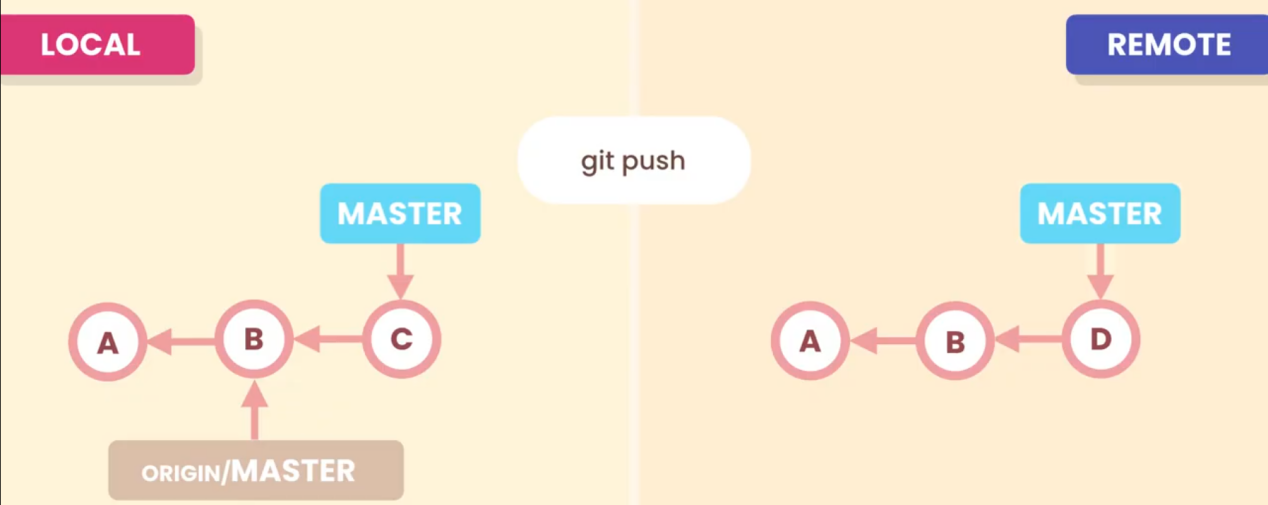
Git push sends our commit to remote repository

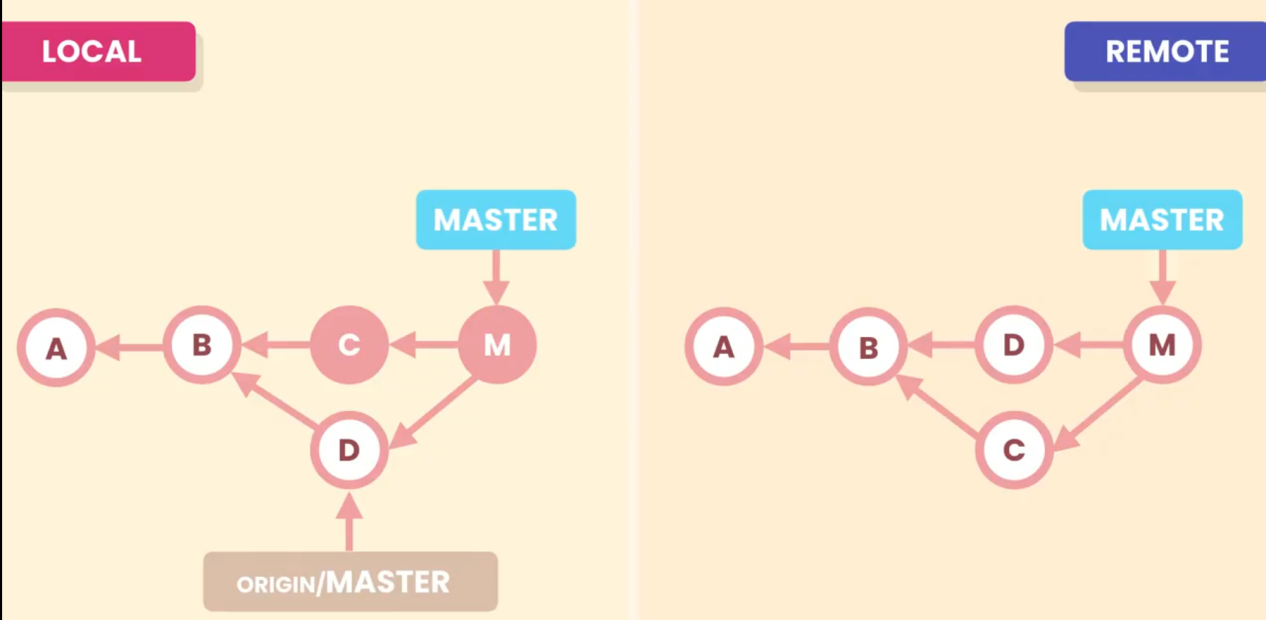
Git push origin master

Origin is the place we wanna push master is the branch we are pushing

For suppose before you pushed someone else pushed and committed so you will not be able to push your commit since both of the git environments are not identical. To get rid of this issue we gotta pull first and then push

If you do git push -f the other persons commit is deleted





Everytime you do git push you have to enter the credentials. To get rid of this tedious task we store these values somewhere and tell git to find em

For mac we use keychain for windows we use windows credential store

Git config --global credential.helper cache # this will strore your credentials for 15 minutes in memory

By default push command doesn’t allow tags to go to remote repository

We have to explicitly push them

Git tag v1.0

If I want to share It with my other team members I have to do

Git push origin v1.0

Git push

To remove a version we do

Git push origin --delete v1.1

Git tag -d v1.0 #to delete the tag from local repositorry

Release Notes:

Release notes to write about the tag version

Sharing branches:

When we create a new branch in our local repository we can’t push it to the main branch bcoz there wont be this new branch in remote repository

Git branch -vv #to check it

So we need to set upstream for the private branch

Git push -u origin feature/change-password

Git branch -r

Collaboration workflow:

Let’s say me and ganesh are wants to work in a branch. We can do that by any of one of creating a git branch and pushing it Into the github or we can creaate a branch in git hub and pull it

Let’s say you created a branch and

After that you fetched a branch.

Git fetch feature/change-password

So you wont be able to see it.

Git branch -r

When we ran fetch we only got a remote tracking branch

So we needs to create a private repository that maches to this remote tracking branch

Git switch -C feature/create-password origin/feature/create-password

To delete the git branch on remote we do

Git push -d origin feature/change-password

To remove the tracking branches that are not under remote branches we use git prune

Git prune origin

Git remote -v

To add a new remote

Git remote add upstream paste\_the\_url

Upstream is a name you can name what ever you want

Git remote rename upstram base

#to rename it

Git remote rm base

#to remove it

==================================================================================

Rewrite the history

Why do we need it?

When the commit msgs are not meaningful and we have large msgs we need to rewrite the history so that we can understand what we actually did before.

If we have small commits then we can squash all of them to make one large commit.

On the other hand if we have large commit we can split it into small halves to make it understandable

Reword commit msgs

We can drop accidental commits