Version control system

Is a soft that tracks and manages changes to files over time
VCS allow to revisit previous versions of file, compare changes between versions
Others - subversion, mercurial

GIt != Github

Git is VCS where as Github is used to host git repository in cloud

Commands

git --version

git config user.name
git config --global user.name “ar_sumit"

git config user.email
git config --global user.email "arorasumit52@gmail.com"

open .
(Opens current directory in window form)

ls folder
(looks in content of folder)

A git repository is just a git workspace which tracks and manages files within a folder
All git repos have their separate histories

git status
(tells about status of repository)

git init
initialize new repository

Using 'master' as the name for the initial branch. This default branch name
is subject to change. To configure the initial branch name to use in all
of your new repositories, which will suppress this warning, call:

git config --global init.defaultBranch <name>

> Names commonly chosen instead of 'master' are 'main', 'trunk' and 'development'. The just-created branch can be renamed via this command:

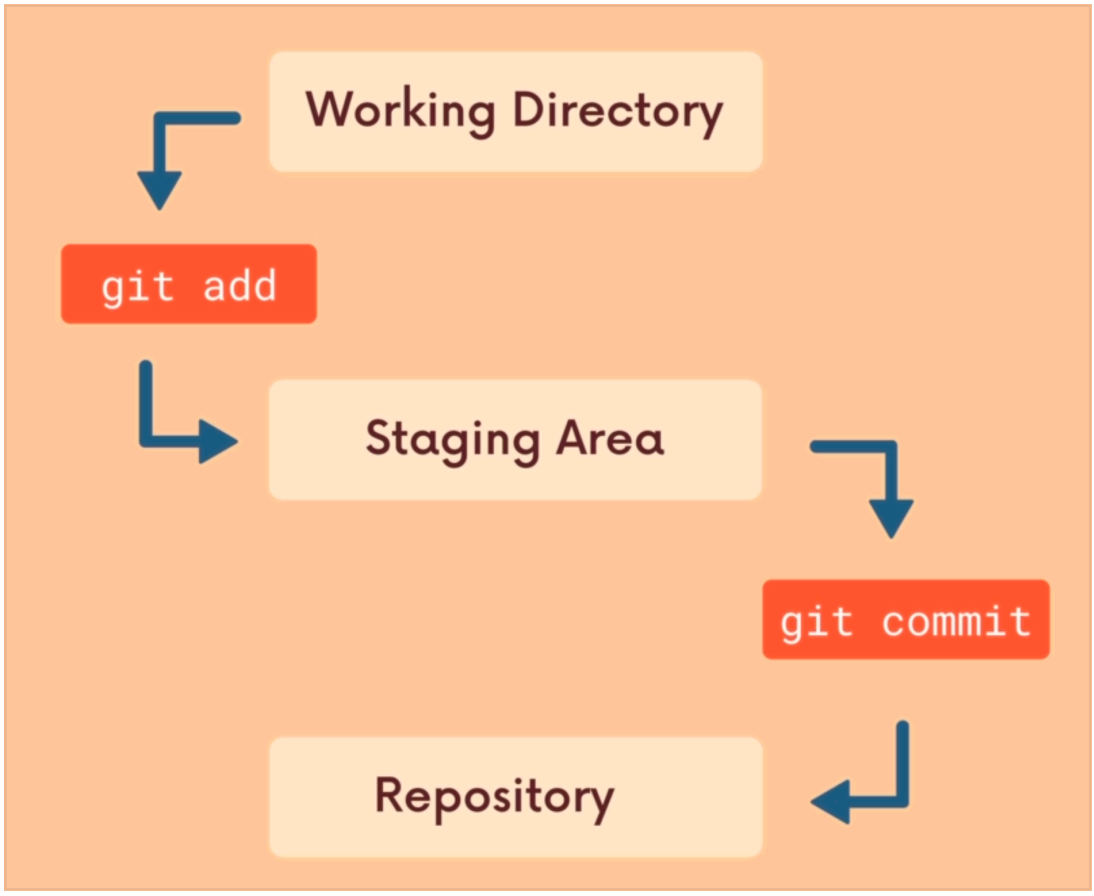
git branch -m <name>

if you remove .git folder all history of the repo is lost basically git hi hat gya

git works nested way that is any folder inside git repo will also be considered as part of git repo only

— don't initialize another repo inside a repo

Committing



commit can be thought of as adding a checkpoint in your projects workflow or snapshot of your project in time

it is not same as saving it is built on top of saving sort of

typical commit is grouping of files that we made changes to

git add file1 file2...
add files to staging area

git add .
git add all files

git add folder/

git commit
used to actually commit files

git commit -m “message”

git commit -a -m “message”

— git doesn’t consider empty folder as worthy of being tracked
— when inside folder it is referred as .\

git config --global core.editor "code --wait"
sets vs code as default text editor

git log --abbrev-commit
shortens the commit hashes

git log --pretty=oneline
pretty displays things in particular format

git log —oneline

Amend Commit just the last one ->

git commit -m “some commit”
(forgets file :( )
git add file
git commit --amend

if you want to change name simply use amend or use along with flag

to modify a file make changes stage the file and amend

We can tell git to ignore a particular file or folder by adding a .**gitignore** file->

.DS_store
*.log
foldername/

Branching->

each commits get a unique hash and it points to at-least one parent commit

- master is the default branch name when you create a git repository

- HEAD is simply a pointer that refers to the current location in the repository, that we are checking out or looking at
- it simply points to a particular brach reference
- or head is a reference to branch pointer and branch pointer is where branch currently is

git branch
gives the list of current branches we have not unless you made even a single commit

git branch <branch-name>
creates new branches but doesn't switches

git branch -v
last commit, tip of the branch

This makes a new branch based on the position of current HEAD

git switch <branch-name>
shift to a particular branch

git checkout <branch-name>

Historically we used git checkout to switch branches
it does lot more than just switching
it restores working tree files

git switch -c <branch-name>

create and switch to branch in one go

git checkout -b <branch-name>
above equivalent

If you work on a branch and checkout another without saving git will not allow change until commit or stash those files

And if you create a file on a branch that file is not on any other branch then instead of error the file will follow you every where

git branch -d <branch-name>
delete the branch, but you should not be on that branch and it should be merged (usually when changes are made on branch it will not merge)

to solve above use

git branch -D <branch-name>
forces to delete

git branch -m “new branch name”
move/rename branches, have to be on the branch

GIT MERGING

branching makes it easier to work in self contained context but, often changes are incorporated from one branch to other

git merge <branch-name>
command to merge

We merge branches not commits
We always merge to current HEAD branch

——————————————— (head -> this branch) **merge into this**
\ /
——————————

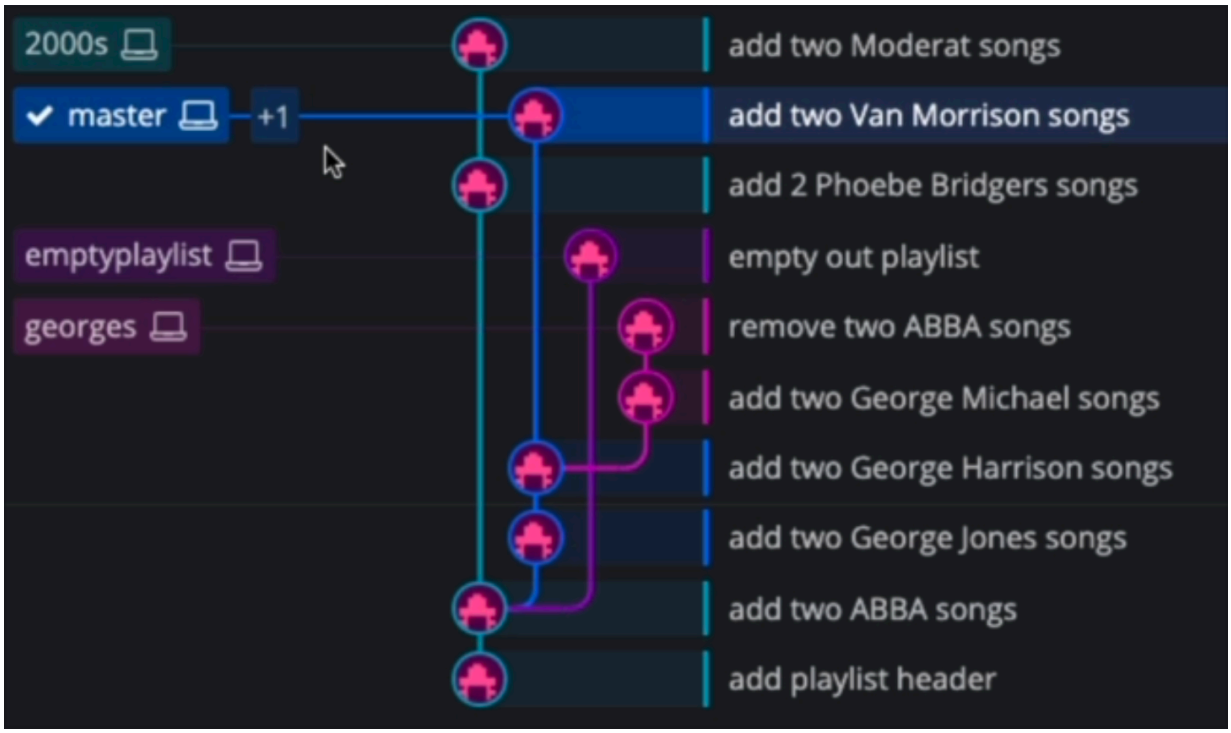
— branches are just defined by a branch pointer the visual picture don’t exist they

are straight line with pointers

Types of merge

1. fast forward merge

master pointer branch pointer
master
——— | |
\ branch -> ———————— -> catch up on additional commits simply on branch
———————|



when case is not as simple as fast forward merge ,
In such conditions arise where git generates a “merge commit”

2. when info is present on master as well (without conflict)

master
——— ——— -> ———————— | (merge commit)

(now we cant simply fast forward as info is lost on master)

Git creates this merge commit for us and asks for a message,
This merge commit is the first commit we have seen that has 2 parent commits

Merge strategy
- ort (faster)
- recursive

Merge Conflicts

- If two people make changes to same file on two different branches or in one branch i delete a file and on another someone modified the file this results in merge conflict

conflict markers

<<<<HEAD - (what we had on branch initially CURRENT CHANGE)
>>>>> <"merge branch name"> (incoming changes)

undoing the merge and going to previous version effectively undoing the merge
git reset --hard <commit-before-merge>

Git DIFF

git diff command is used to view changes between commits, branches, file , working directory and more
informative command

git diff

without additional options diff lists all the changes in **working directory that are not staged** for next commit
i.e it compares **staging area and working directory** (all un-staged changes)

git diff workflow

1. it lists the files it is comparing usually same file's different version, it declares one as A other as B
2. index line is the meta data about the files being compared
3. file A and B each are assigned a symbol

file A gets a minus -
file B gets a Plus +

- - - a/index.html
- - - b/index.html

chunks

git doesn't show entire contents of a file but only the portions or chunks that were modified

chunk header

each chunk starts with a header found between @@ and @@
@@ -3,4 +3,5 @@

- for file A , + for file B
- from file A 4 lines are extracted starting from line 3
- from file B 5 lines are extracted starting from line 3

git diff HEAD

list all changes in the working tree since last commit (includes all staged and un-staged changes)

when previous file doesn't exist before A file is /dev/null

git diff --staged or --cached

will list changes between last commit and staging area

git diff HEAD [filename] or git diff —staged [filename1] [filename2]..

we can view changes within a specific file by providing git diff with filename

git diff branch1..branch2

git diff commit1..commit2

◯ TO refer to the parent of use **HEAD~1**

STASHING

when we want to change branch but we have unsaved work

1. git wont allow to switch
2. the file will follow you to the branch you want to change to (assuming there are no conflicts)

Stashing deals with this problem so you don't have to wait till your work is done

git provides stashing so that you can stash these uncommitted changes and return to them later without making
unnecessary commits

git stash or git stash save

stashes changes, when run it take all uncommitted changes staged and un-staged reverting the changes in your working copy
but changes are present and can be retrieved

git stash pop

to remove the most recently stashed changes in your stash and reapply to your working copy (wherever i may be
different branches as well)

git stash apply

to apply whatever is stashed away without removing it from the stash, can be useful when want to apply changes on

multiple branches

You can put multiple items into stash

git stash list
view all stashes (WIP: work in progress)

git usually refers to the most recent stash when you use git stash apply but we can reference the stash using stash id

git stash apply stash@{2}

when applying multiple stashes you may run into errors just clear the entire file and you are good to go apply changes from
another stash

git stash drop stash@{2}
to drop a particular stash

git stash clear
clears entire stash

———

git checkout <commit-hash>
to view a previous commit

we only need first 7 digits of the commit hash

Detached head?

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

head ->branch reference -> commit

that is to say when new commits are made branch reference updates but head still points to branch reference
when we change branches head point to new branch reference therefore

head points to branch not the commits

now what we checking out a commit HEAD points to the commit instead of branch pointer
(we are not on a branch we are a detached head)

usually cd .git> cat HEAD -> refs/heads/master ; inside each branch name we have the hash i.e where branch pointer is or branch reference

is detached head mode

cd .git> cat HEAD -> cae0236c4dad11414ab0b15321bb6301be02c1a4

when in detached head mode you can

- examine contents of old commits, view files etc
- reattach the head
- create a new branch and switch to it, you can now make and save changes since head is no longer detached

To reattach simply switch to that branch i.e

git switch master re attaches head to master

we can also checkout by :

git checkout HEAD~1
view one commit before head (parent)

git checkout HEAD~2
view two commit before head (grandparent)

git switch -
takes you back to where you were before detached head

discarding changes to files:

1. using checkout

git checkout HEAD <file-names> or git checkout -- <file-names>

it is different from just viewing
using checkout to discard changes to a file or to make it what it looked like when you last committed it, or simply reverting to HEAD

---

2. using restore

git restore is a new command that is used to undoing changes as checkout does a million things takes

discard changes since last commit / HEAD

git restore <filename>
same as checkout seen above

to restore contents from a particular commit use the --source flag

git restore --source HEAD~1 <filename>

using restore to un-stage files :

git restore --staged <filename>
un-stage the files

---

3. using reset

git reset <commit-hash>

removes commits not the changes themselves

git reset --hard <commit-hash>

removes commits and the changes themselves

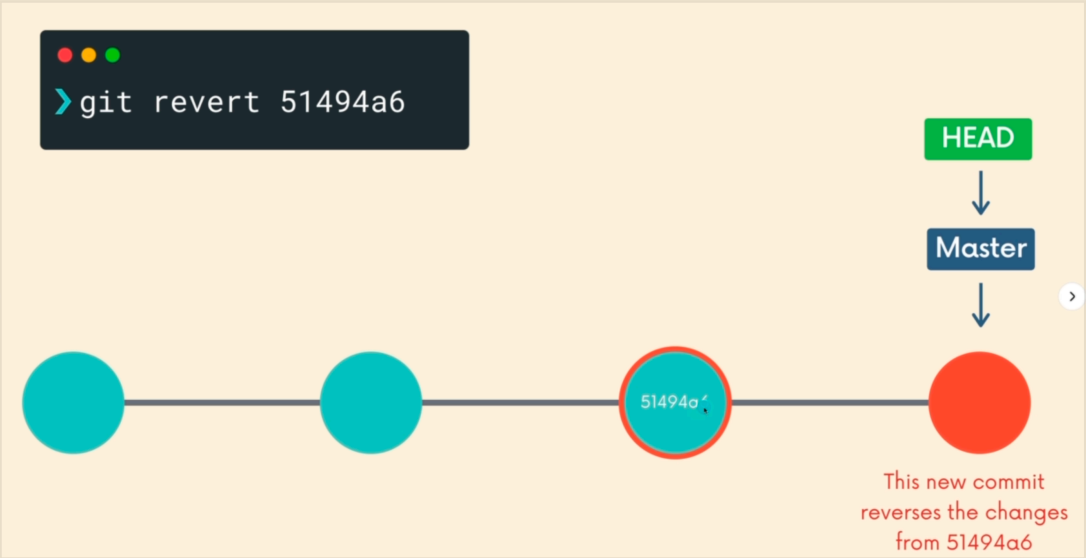
4. using revert

git revert <commit-hash>

it is similar to git reset in terms it undos changes but

-git reset moves the HEAD pointer back eliminating changes

-git revert instead creates a brand new commit which reverses/undos the changes from a commit
the commit still exist in the history but changes are reverted



==========

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

Github

Github is a hosting platform for git repositories, it helps people share and collaborate on repos

cloning

allows us to get a local copy of a preexisting repository
git clone is part of git not github i.e. you can clone from other platforms as well

git clone <url>

git will retrieve all files associated with repo and copy to your local machine; and will create a new repo with full git history

don’t clone inside a repo

SSH KEYS

you need to be authenticated on github to do certain op like pushing from local machine
Your terminal will ask id/pass everytime to bypass set an SSH key which can connect to github without pass

Putting projects on github

1. Existing repo locally

create a new repo on github
connect your local repo (add a remote)
Push up your changes to git hub

2. Start from scratch

create brand new repo

clone locally , do some work , push changes

REMOTE

before we push anything on github we need to tell git about our remote repo on github
i.e set push up destination

in git we refer these destinations as remote
each remote is simply a url where hosted repo lives

git remote or git remote -v
tells about any existing remote for repo

when you clone there is an existing remote setup automatically, usually the url from where repo is cloned from

git remote add <name> <url>
remote composes of label and a url

git remote add origin https://blah blah

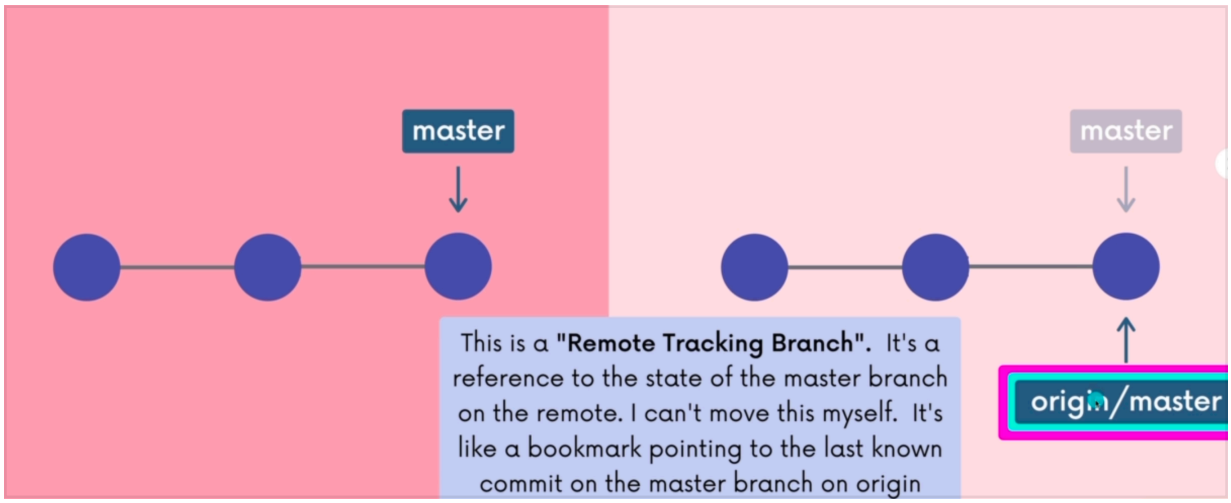
origin -> standard name used (telling git every time i use origin i mean this url)

git remote rename <old> <new>
git remote remove <name>

Git PUSH

git push <remote> <branch>
git push origin master

git push -u origin master
what -u (--set-upstream) does is it sets up the upstream for branch so we don't need to add above command as it establish the link
b/w branch as i have to push changes to it so simply git push will work, for each branch set individually.



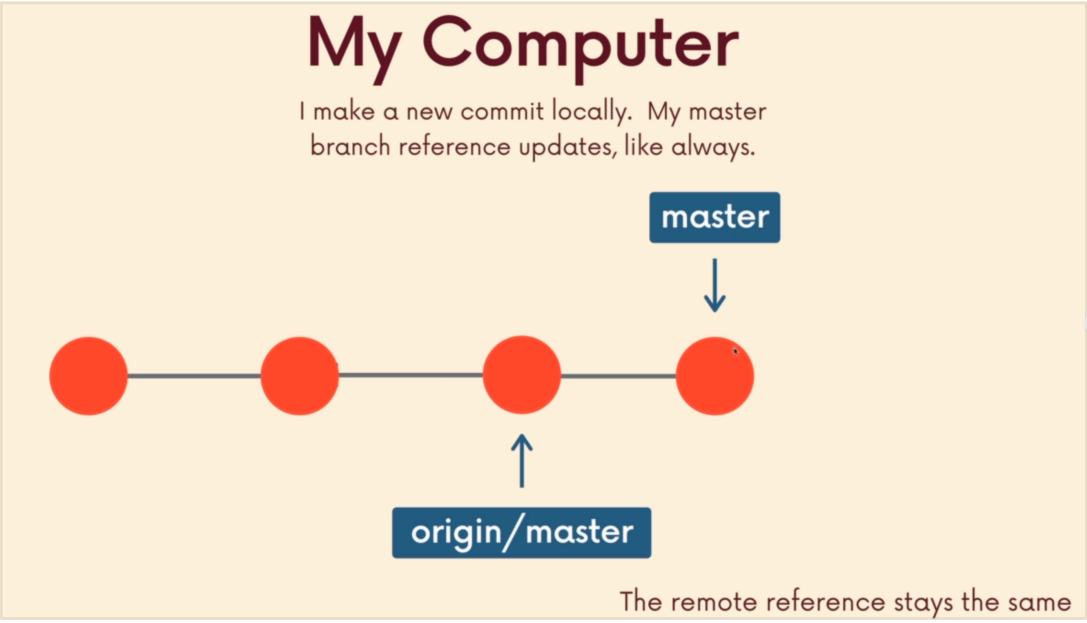
previously we only had master branch now that we clone shit into our own machine we have another branch reference called
remote tracking branch which tells state of branch on remote, it is like a bookmark pointing to last known commit on branch on origin

at the time you last communicated with this remote repository here is where “x” was pointing

pattern : <remote>/<branch>

remote branches :

git branch -r
to view the remote branches our local repo knows about



to view when you cloned repo or origin/master use

git checkout origin/main

when you clone a repo you have the project history till that moment of time but when you put git branch command it only shows main
so what is happening? , but if we run git branch -r we get all the remote branches i.e. it knows where the other branches on origin are pointing but in my workspace only master is their

so when you clone there is link b/w main and origin main as we saw (you didnt create this)

you can checkout branches origin/food in detached head but i want own branch of food connected to origin/food

use

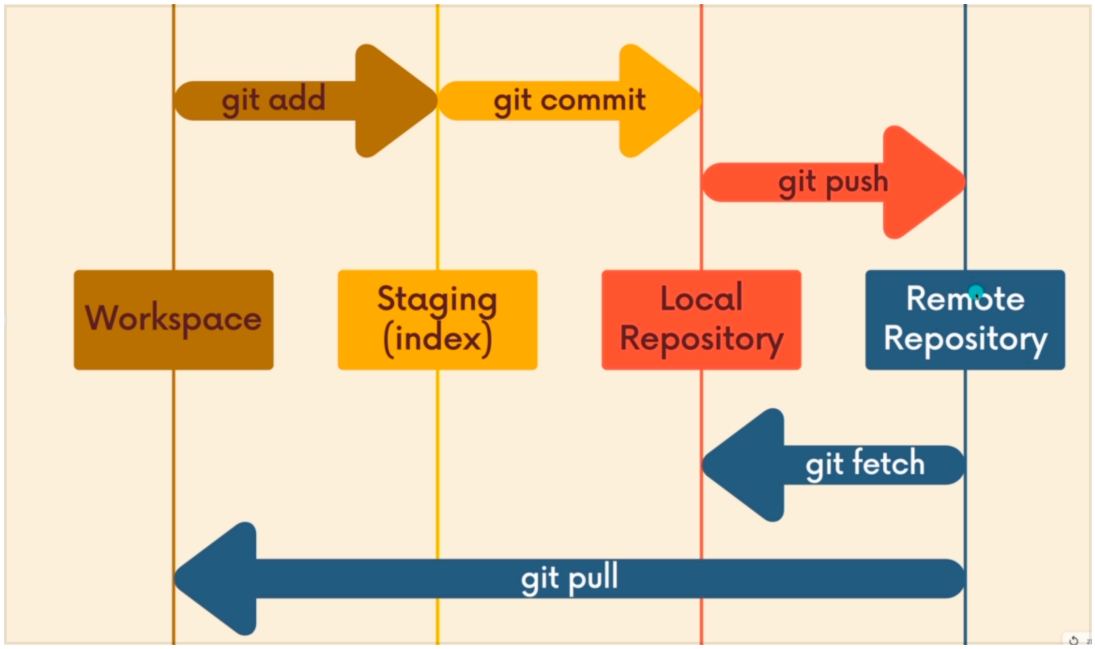
git switch <puppies> or git checkout --track origin/puppies

it sets up a local branch and will set it up to track the remote branch

- for a current branch to setup a remote upstream use:
- **git branch --set-upstream-to origin/pulling-branch**

FETCH AND PULL

while i was working locally someone just made 3 commits on remote branch but i dont have those changes what do i do heres why fetch and pull are used



Fetching

fetching allows us to download changes from remote repo but these changes will not be integrated into our working files
it lets you see what others have been working on without merging changes into local repo

git fetch <remote> / <remote> <branch>
fetches branches and changes from a specific remote repository, it only updates remote tracking branches

in short your origin/main is updated and local main is untouched

Pulling

git pull is similar to fetch but actually updates our HEAD branch with whatever changes are retrieved from remote

pull = fetch + merge

git pull <remote> <branch>
whatever branch we run it from changes will be merged into

A good practice is to pull before pushing changes

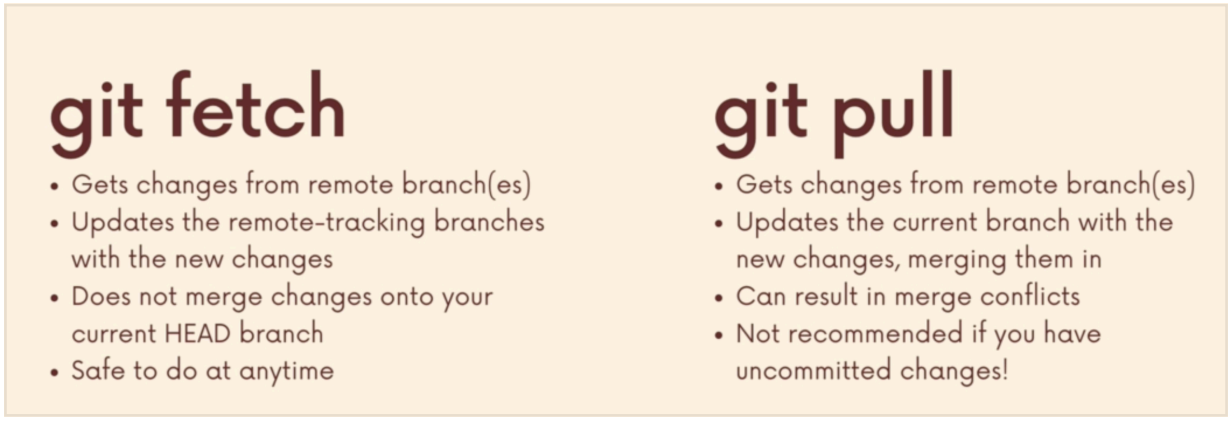
You have divergent branches and need to specify how to reconcile them.
You can do so by running one of the following commands sometime before
your next pull:
hint:

git config pull.rebase false # merge
git config pull.rebase true   # rebase
git config pull.ff only   # fast-forward only

You can replace "git config" with "git config --global" to set a default
preference for all repositories. You can also pass --rebase, --no-rebase,
or --ff-only on the command line to override the configured default per
invocation.

if we run git pull (like this only)

git assumes remote to origin
branch will default to whatever tracking connection if configured to your current branch



———

REPOS

public:
private: owner and people given access to

Readme file:

is used to communicate what project does, how to run the project, why it is noteworthy, who maintains the project

if readme in root of directory github auto displays it in home page
.md -> markdown files

———

MARKDOWN

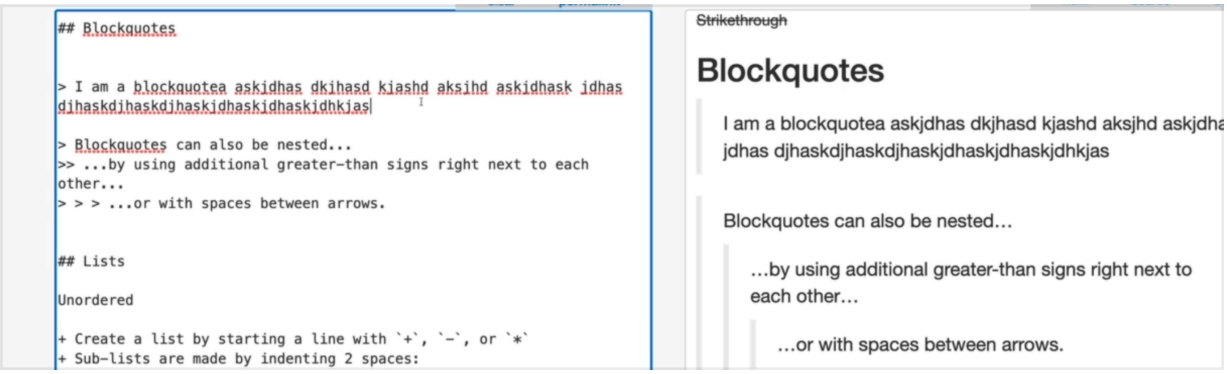
h1 Heading
h2 Heading
h3 Heading

Horizontal Rules
___
***

abdhkc -> bolds the text
italic

~strikethorugh~

Blockquotes



Lists

create list using +, - , *

highlight code `code`

```
block of code
```

for syntax highlight append js etc at end of first 3 backtick

Links

[Link text](urls)

Images

![name image](url)

Github Gists are a way to share code snippets and simple fragments with others, Gists are easier to create but have far less features

Github Pages are webpages that are hosted and published via github. These are static pages so no server side code
only H/C/J

This comes in two variety User site and project site

User site : one per github account used for personal site
username.github.io

Project site: unlimited, each repo can have a project site
username.github.io/repo_name

To set it up

it looks for index.html page
use gh-pages conventional, set from settings where you want git hub pages set up

|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

WORKFLOWS:

<u>Centralized workflows:</u>

everyone works on the master/main

<u>Feature branches</u>

branches are feature oriented, work on individual features alone

At some point all the feature branches have to be merged into the main so how we do it matters

- do at will
- communicate with team some how
- **PULL REQUESTS**

PULL REQUESTS

Pull request are native to git hub and not git, they allow other devs to alert the team about new work that needs to be reviewed
They provide a mechanism to accept or reject the work on a gi ven branch

Do some work locally
push up the branch
open PR using the feature branch just pushed to github
wait for PR to be approved and merged

PR conflicts

Step 1: From your project repository, bring in the changes and test.

```
git fetch origin
git checkout -b new-heading origin/new-heading
git merge main
```

Step 2: Merge the changes and update on GitHub.

```
git checkout main
git merge --no-ff new-heading
git push origin main
```

—no --ff -> telling git to not do a ff

update:

simply pushing the branch onto which main is merged and conflict is resolved will work as well now :)

BRANCH PROTECTION RULES:

rules to prevent forced pushing , prevent branches from being deleted and may require status check before merging

Fork and clone workflow

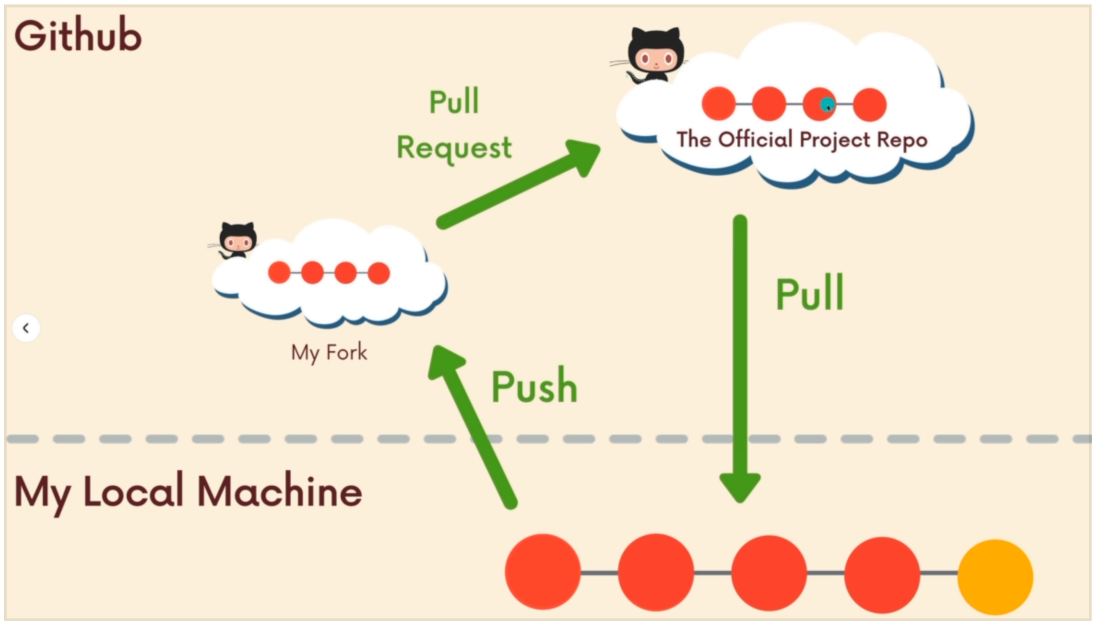
Forking allows us to create personal copies of other people's repository, we call these copies fork
forks give you all rights of the repo copy

when cloned remote is setup for you automatically

clone this repo and work on it

further we have to setup another remote which allows us to get updates from the original repo

called upstream or original conventionally

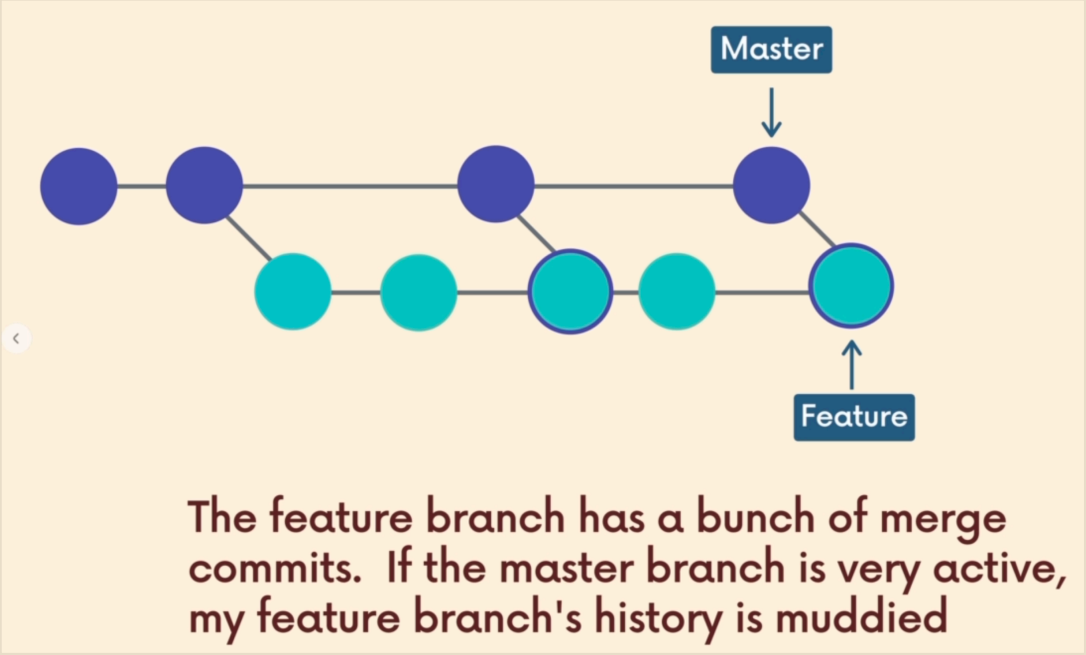


```
Stevie ❯ cd fork-and-clone
fork-and-clone ❯ git log                    main
fork-and-clone ❯ git remote -v              main
origin  git@github.com:StevieChicks/fork-and-clone.git (fetch)
origin  git@github.com:StevieChicks/fork-and-clone.git (push)
fork-and-clone ❯ git remote add upstream https://github.com/Col
t/fork-and-clone.git
fork-and-clone ❯                                                main
```

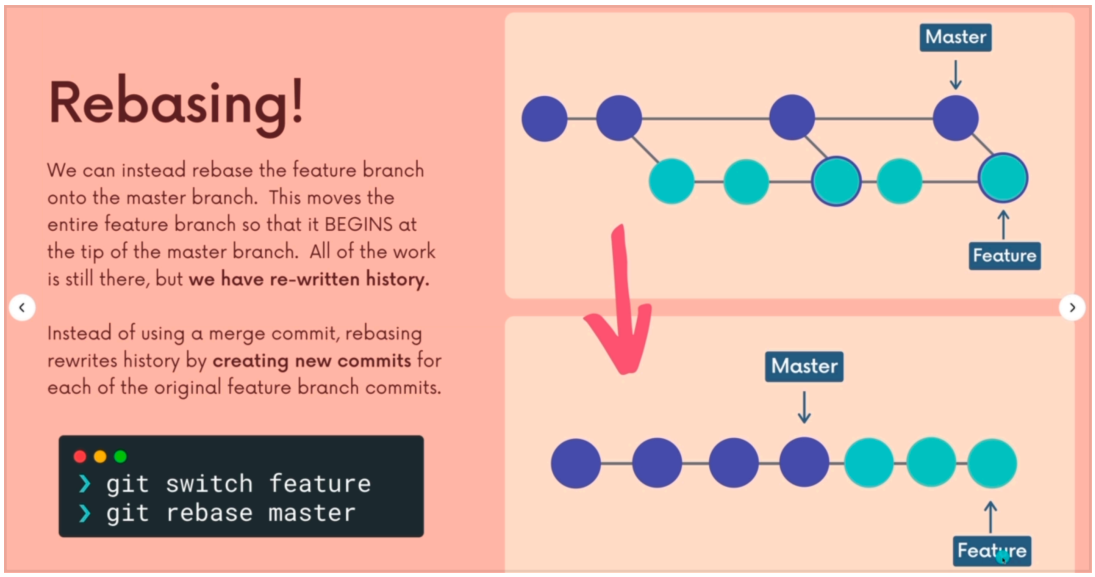
REBASING

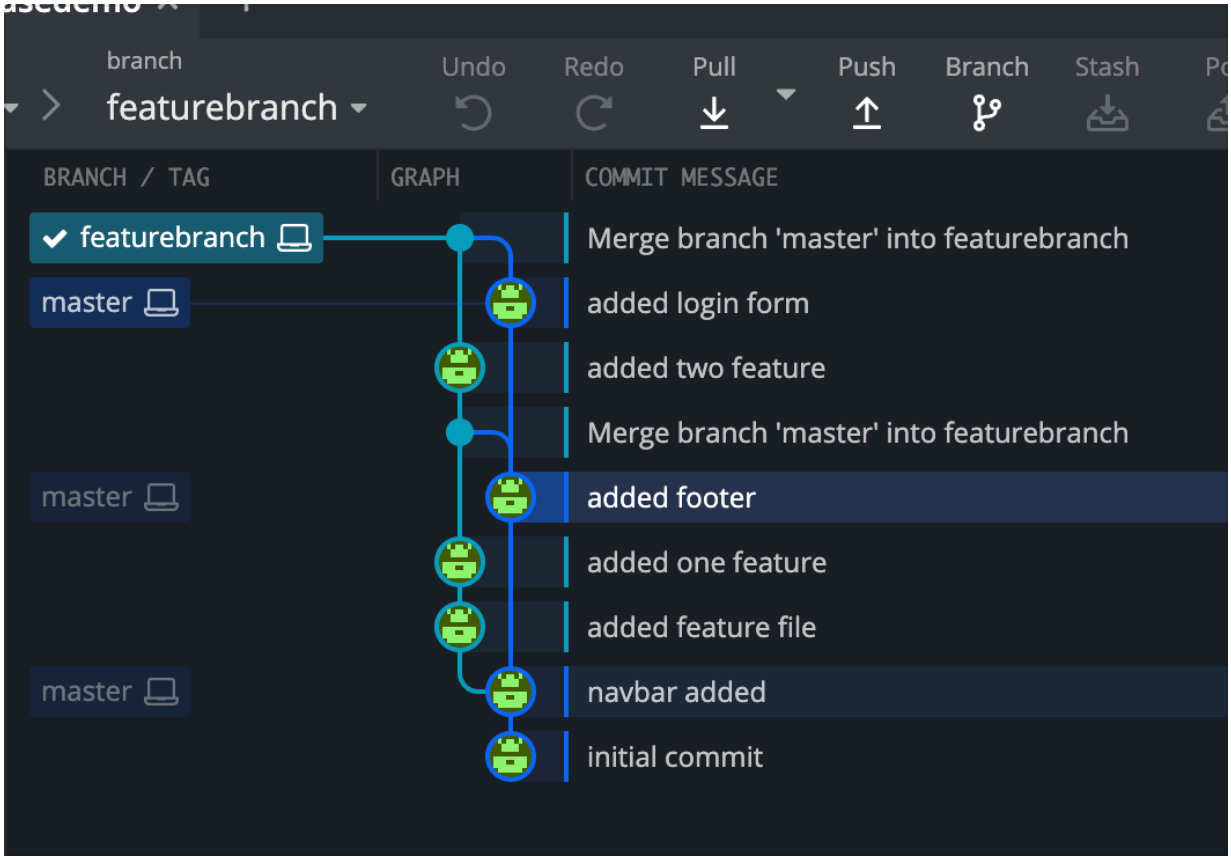
- Two ways to use rebase

Alternative to merging
as a cleanup tool



rebasing can help us with when we have history that is muddies by frequent merging
When we Rebase we are re writing history

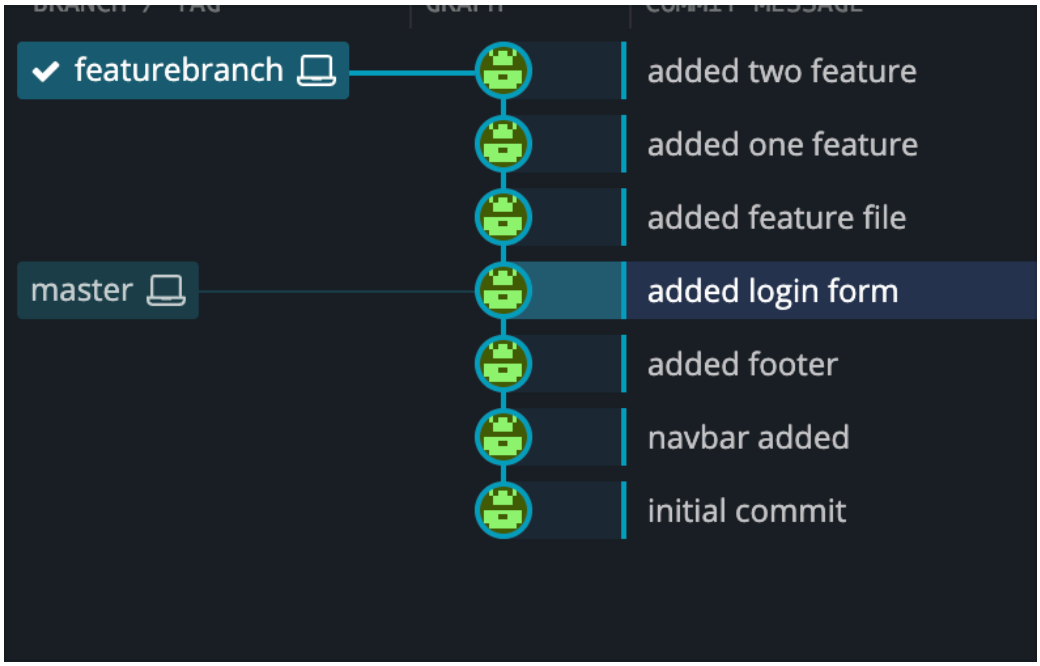


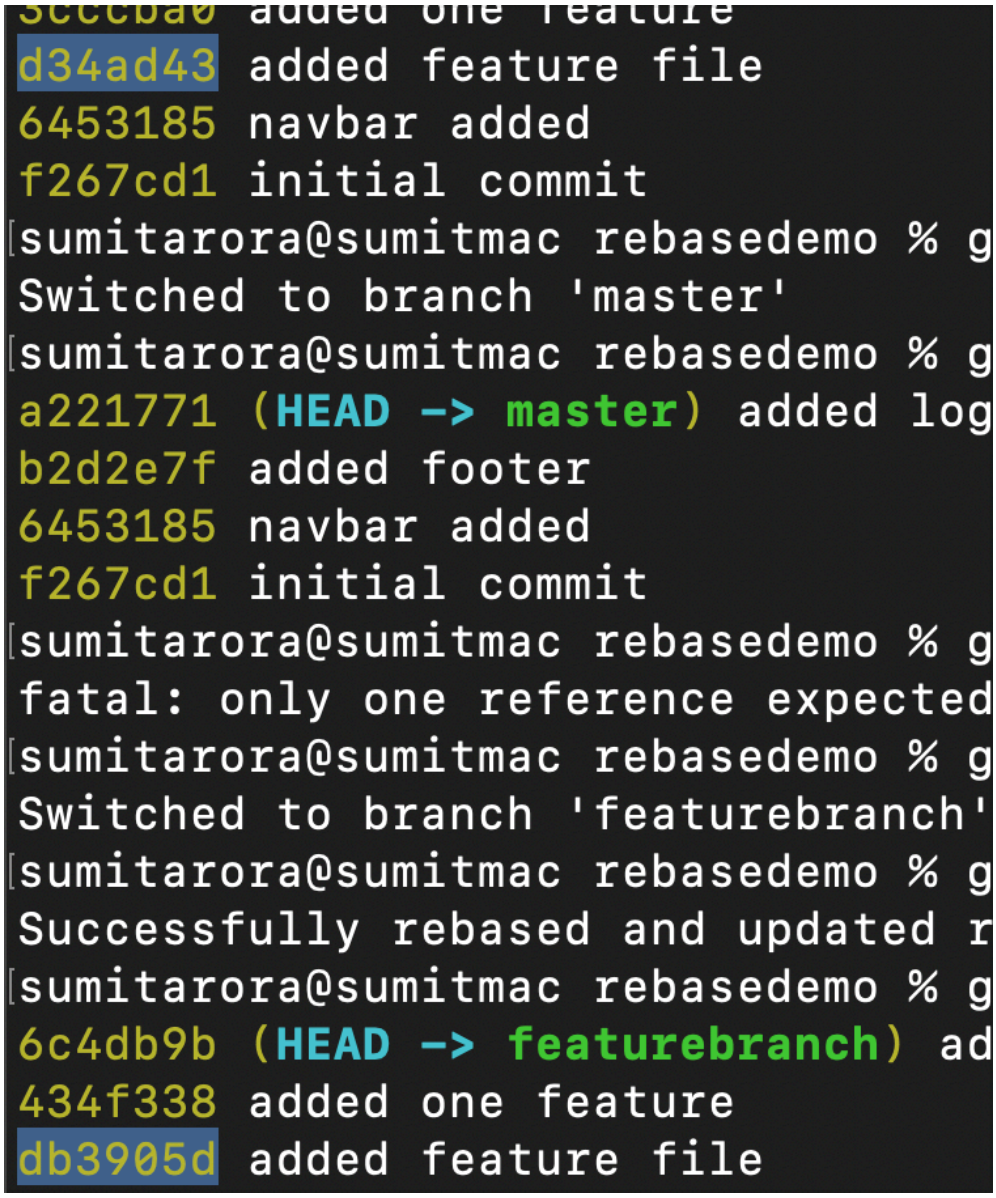


when you rebase instead of screwing master branch what it does is it takes feature branch make new commits on the base of old ones and put it at tip of master branch

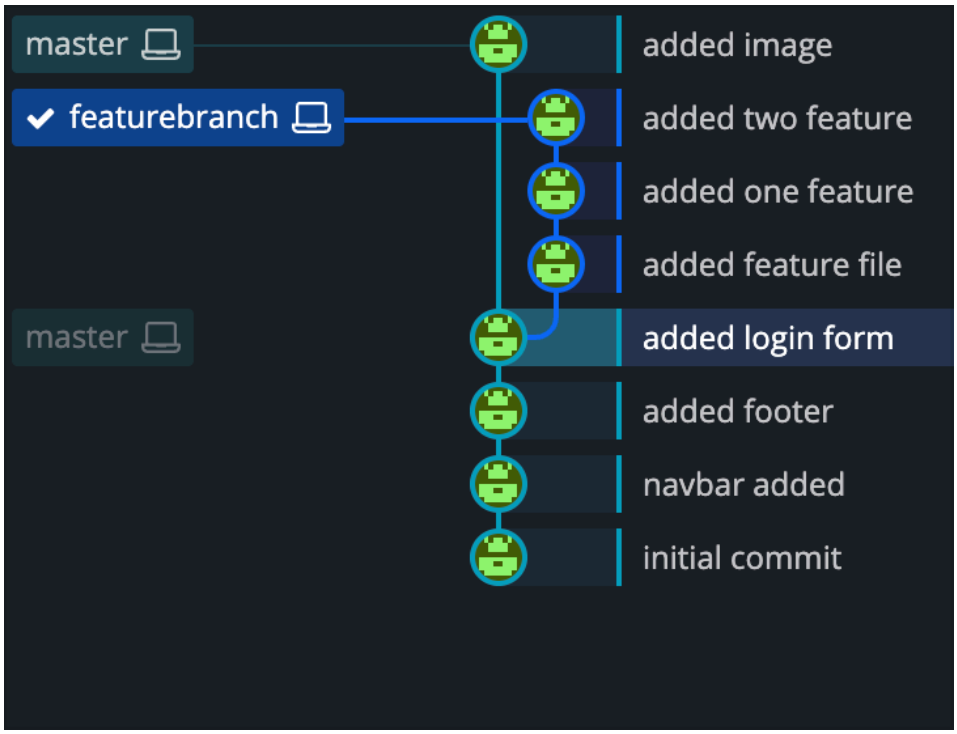
git rebase <master>

name of branch you want to rebase your branch onto

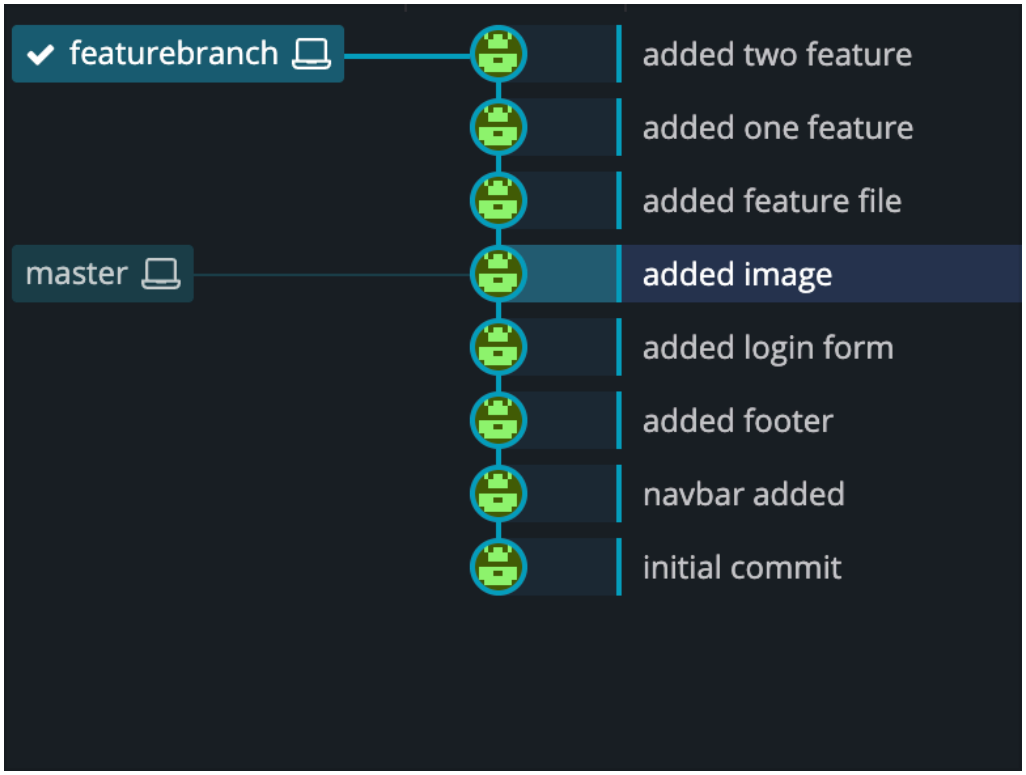




commit hashes changed indicating these are new commits



Instead of merging you can simply rebase



Since rebasing rewrites history, never rebase commits that have been shared with

others i.e. if you have already pushed to github do not rebase unless you are sure that no one else on team is using those commits

When conflicts occur partial rebase can happen
to go back use

git rebase --abort

Resolve all conflicts manually, mark them as resolved with

git add/rm <conflicted_files>,

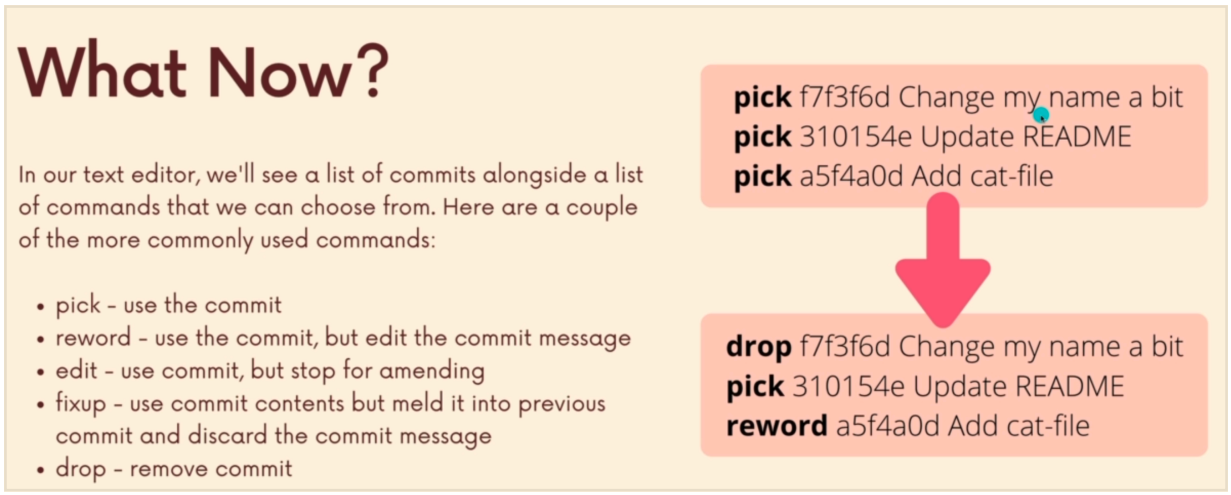
then run **git rebase --continue**

You can instead skip this commit: run "git rebase --skip".

Another way to use git rebase is when we want to rewrite, delete rename or reorder commits(before sharing them)
i.e using it as cleanup tool

to do this we enter interactive mode with -i flag Also we want to specify how back we want to rewrite commits
also we are not rebasing onto other branch instead we are rebasing a series of commits onto HEAD they are currently based on

git rebase -i HEAD~4



if you change till ~9 even if one is changed all others are regenerated and new

GIT TAGS

Tags are pointer which refer to particular point in git history, we can mark a particular moment in time with tags
They are used to mark version releases
They can be thought of branch references that do not change, it refers to just that commit as a label

Two type of tags

lighweight - name
annotated tags - have meta data author etc

Semantic versioning -> how version are encoded
Major minor patches
X.Y.Z

git tag
view tag

git tag -l "*beta*"
filter tags

git checkout mytag
checkout tags like detached head, other way of reaching commit

git diff tag1 tag2

git tag <tagname> -> lightweight tag
by default it will point to where head is pointing

git tag -a <tagname> -> annotated tag
by default it will point to where head is pointing

git show <tagname>
to view who made it use

git tag <tagname> <commit> -f
force tag to new commit or move

git tag mytag commit
add tag

git tag -d <tagname>
delete tag

by default when pushing to remote repo tags arent pushed

use **git push --tags**

OBJECTS FOLDER

This is the folder where all our repo file lives, this is where git stores backup commits etc

REF LOGS

git keeps records when the tip of branches and other references were updated in the repo we can view this using

git reflog command

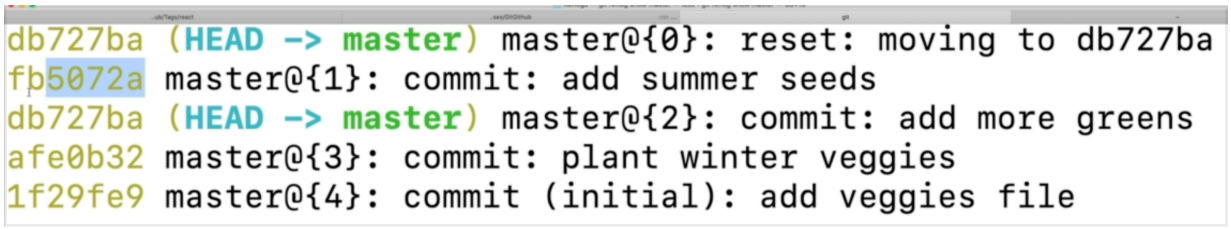
reflog are local and expire aftr 90 days

git reflog show HEAD or branch name

HEAD@{01} - > most recent

RESTORE LOST COMMITS

if you reset and lose commit we can use reflog



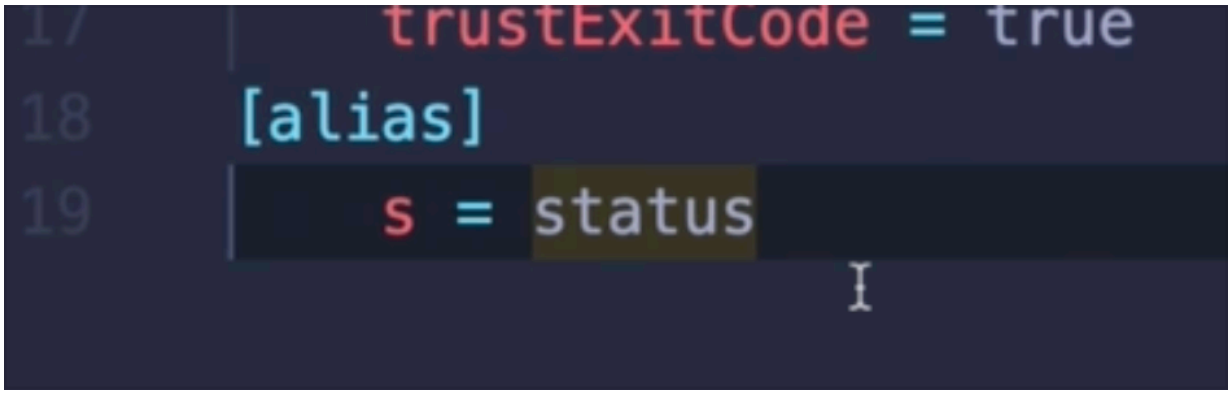
git reset —hard master@{1}

GIT GLOBAL CONFIG FILE

git looks for global config file in ~/.gitconfig or ~/.config/git/config

GIT ALIAS

it is just shortcuts for cmds



for alias use

cm = commit -m (it will append any args passed)