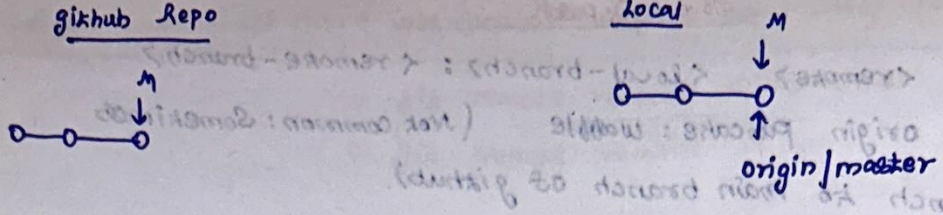


origin/master (origin/main) : Reference to state of master branch of the remove.
(default) Can't move until fetch/pull

Masteen: moves at each commit

Bookmark pointing last known ref commit on masteen (Based on origin)!

At the point last communicated: what's the state of repo.

origin/master

upstream/logoRedesign — custom remote name
(upstream - common remote name)

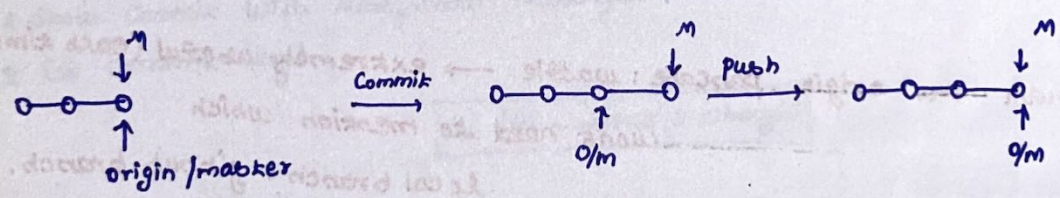
---

git branch -r → origin/master

View the remote branches — local repo knows about

hold my spot Kind of thing!

Checkout Remote tracking branches



* git status (your branch ahead by 1 commit) — push (Compares M & O/m)

note: Until push: Remote reference (origin/masteen) won't move.

Say: I am not doing push

what will be — repo — when I cloned

git checkout origin/master

git branch -r

origin/main

Remote branches

git branch → But github shows

* master

4 branches there!

Cloning: All data & history (doesn't mean all files in my workspace)

What's going on?:

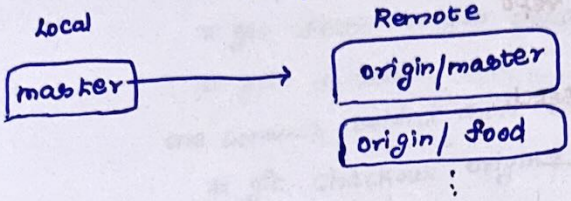
(remote branches)

```
# git branch -r
origin/fantasy
origin/food
origin/main
origin/movies
```

→ knows about all branches & has remote preference.

Full fledged branch (vs) Remote tracking branch:

* default: master is the full fledged branch



What I need to do: create a local branch named 'puppies', connect it to origin/puppies (like master does default).

git origin/food using checkout

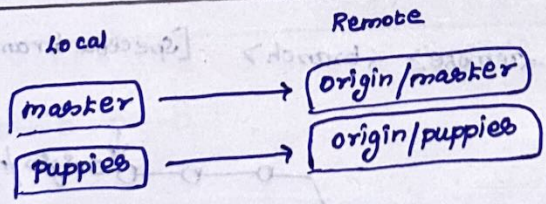
→ git checkout origin/food [puts in detatched HeadState]

Rather than checkingout

1. make one locally, connect it!

```
# git switch puppies
```

[easy: Create one – already there: git does the job Connects].



* we can't switch to branches – already not there (without creating) (make it local – if globally there)!

git switch puppies does the job.

Also we can use (Complicated): git checkout --track origin/puppies

---

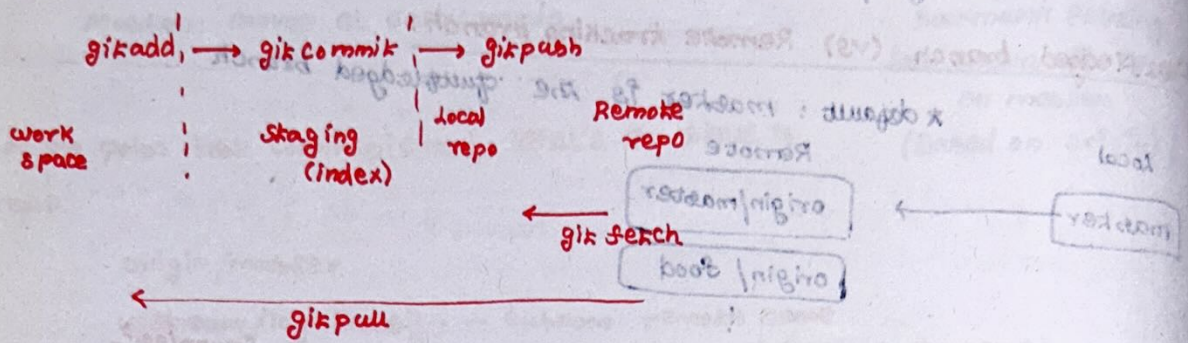
Github — **Git fetch**

(Teammate pushed something)

Local

O/m Master

How to get those changes



git fetch:

* Remote changes to local repo (not workspace)
* won't integrated into our working files
* See: what others working without merging into local repo

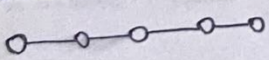
'please go and get the latest info from github — but don't screw up my working directory'

git fetch <remote>

fetches branches & history — from specific remote repo — update remote tracking branches.

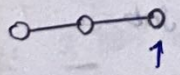
git fetch <remote> <branch> [Specific branch]

Github:

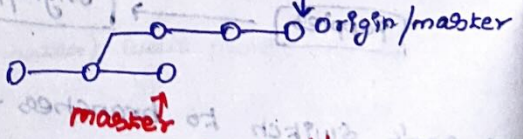


fetch remote

Local:



master



demo: git fetch

github: added 2 files in 2 branches

(default: origin)

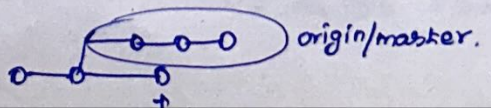
git switch movies

git fetch origin

git status

git fetch

Your b is behind origin / movies by 1 commit [use pull]



git checkout origin/movies

ls

3 files (added one also showing) – new work (github)

git switch movies

git switch food (1st time: so automatically made connection)

3 files (inc added one)!

why?: made new connection now only! (Remote reference made connection)

---

now add a file to food (github)

git fetch origin food

git status

one commit behind than origin/food.

git checkout origin/food

ls

view file

Git Pull

★ updates working directory! (HEAD branch)

git pull <remote> <branch>

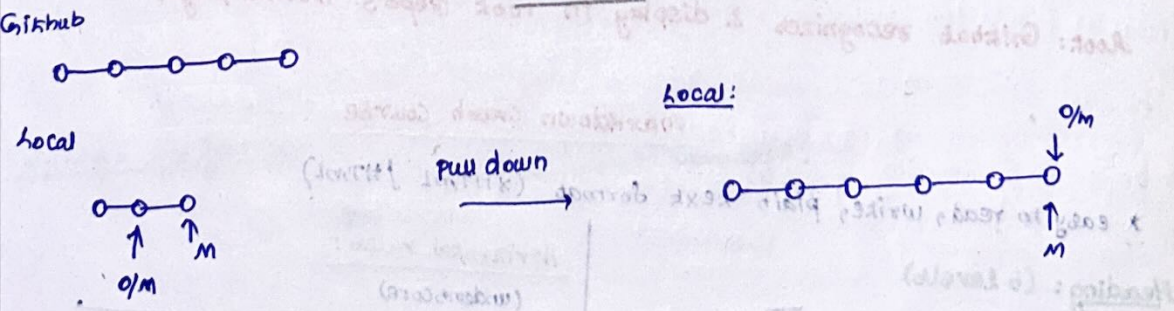
git pull = fetch + merge

Note: Current branch important – where merge happens. (merge conflicts)

git pull origin movies [(movies branch)]

No conflicts: merged

Conflicts



Sometimes: Fast forward

Auto merge

Conflict

food: add coffee.txt → commit

git pull origin food

Conflict: since github also: edited that file.

Resolve Conflicts

push up to github! (to make easy for others).

git push origin food

Shortcut Syntax

git pull

why default: origin — manually we can change (let it be)

branch: current branch! (I'm not gonna pull other branches)

Github: odds & Ends

Public (vs) private repo: (visibility)

Public: Anyone can see - access - clone

Private: owner & access granted ones can clone!

owner alone change (Settings)

Add Collaborators:

1. Settings → manage access → Invite a collaborator

For collaborators: No settings (editable)

Collaborator: can push code!

README.md

1. Communicate info about repo
   - ★ project does
   - ★ how to use (run)
   - ★ note worthy (why?)
   - ★ who maintains the project

Note: Github recognizes & display in root (repo's home page)

Markdown Crash Course

★ easy to read, write, plain text format (xHTML / HTML)

Heading: (6 levels)

⋮

] headings

Horizontal rules:

(underscore) ___

(hyphen) ---

***

Symbols

(c) - ©
(C) - ©
(r)] ®
(R)
(tm)] ™
(TM)
(p) (P) §

+- → ±

** text ** → Bold

__ text __ → Bold

* text * → Italic

_ text _ → Italic

~~ strike ~~ → ~~strikethrough~~

Blockquotes

> text .....

>> text...

>>> text....

Lists:

unordered:

* first
* second
* third
  * sublist → 2space Indentation.

ordered:

1. first
2. second
3. third

(backtick)

`code` → `code`

Code block

```
(or) 4 space Indentation.
```

```
```

Syntax highlighting:

```js → put which language!
```

```
```

Tables

Links → [link text] (link)

Images → ![image text] (img link)

Emoji

Footnote

Superscript

Subscript

definitions

abbreviations.

Paper

doc

README to a project

* github gists: (Share Code Snippets)
* gist.github.com ⟶ public/private

---

* github pages: public webpages – hosted & published via github
* Host a website : static web pages (no server side languages) – JS, HTML, CSS
* eg: fakeer docs, 2048AI,

page site : for every repo – we can have hosted website
every project – Corr website.

username.github.io/repo-name

user site : host – portfolio site/personal website.
username.github.io

---

github pages

Settings ⟶ Github pages ⟶ Select Source (enable) ⟶ Select branch
folder: where index.html file!

Git collaboration

Centralized workflow:

1. Everyone works on master/main (truth)
2. Simple – huge team (problem) – error prone

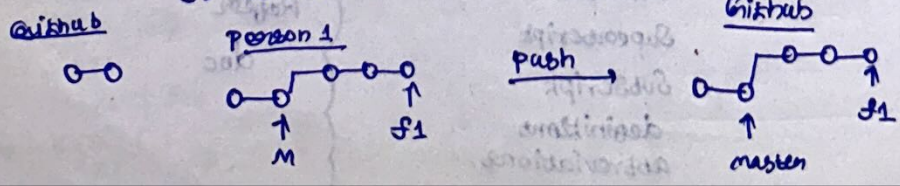
* Some one changes: pushes ⟶ everyone pull (rectify conflicts) ⟶ then push
* Need to pull a lot.

1. Time waste: resolve conflicts & merge code
2. Disturbs others – add new/experiment – trouble.
3. Incomplete pushing – broken code (Some one) (others too)

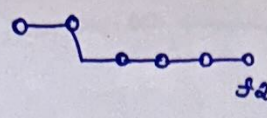
Problem: local changes overwritten by merge!

Feature workflow – common!

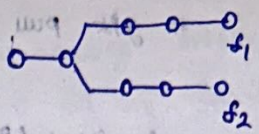
1. Work on branches! (nobody works on main)
2. Treat main – project history (collaborate & share) – without polluting
3. No broken code to main – unless someone messes up!



Person 2

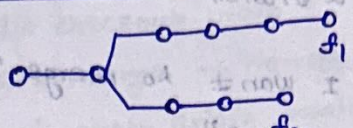


Person 1 asks to check



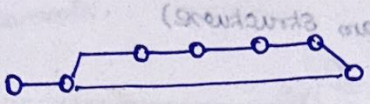
pull down f1 : Verify / look around (add own commits)

Person 2 adds commit & pushes to Github:

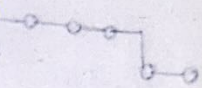


person 1 fetches : checks : pull down the feature!

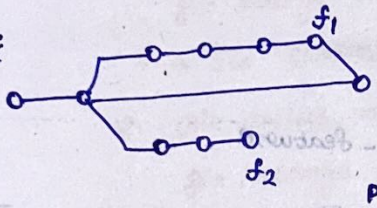
decides : merge to master



Big Company : code review, testing! (code approval)



Github:



person 1	person 2
# git switch -c navbar	# git pull origin main
# Add navbar (html)	# git switch -c pricingtable
# push	# add html & push

merge to main & delete (Not 1000s of branches) will be there!

navbar not working - person2 checkingout.

```
# git fetch origin
# git remote -r
origin/navbar
# git checkout origin/navbar
# git switch - (previous) : pricingtable
# git switch navbar [connected]
# edit navbar, commit, push
# git switch pricingtable ⟶ continue work!
```

Merge feature branches

1. Merge at will (not common) :
2. Send email / chat — discuss & pull requests.

```
# git switch merge
# git merge pricingtable ⟶ Fast forward merge.
# git push origin main      (delete branch)
```

Person 1 : # git pull origin navbar

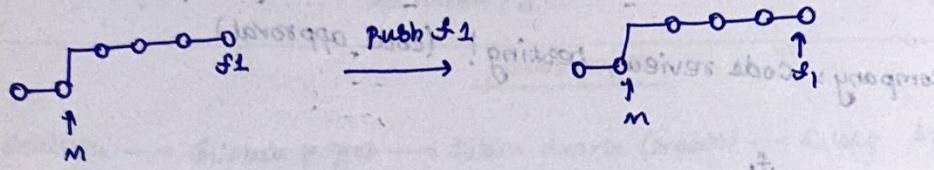
Pull request:

1. Feature in github / Bitbucket
2. Allow members / dev - desk team-members to new work reads review.
3. Approve / reject a branch - discuss / feedback.

"I have this new stuff I want to merge into the master branch... what do you all think about PR?"

Workflow:

1) Companies: main branch protected (can't push code)
2) Create PR (depends on team structure)



Go to branch: compare & pull request

base: master compare: my-new-feature

Write: what is my request, what I've done! [depends on company]

Team-members: Review → Comment

"Conversation"

✓ Merge pull request

Demo

* Compare / pull request → Create PR → merge (no conflict)

Conflicts (Resolve):

* Fix & merge → [github's interactive editor]

git switch -c newheading
change index.html (edit heading etc..)
commit, push branch
#

Can't automatically merge → Conflict there

1) Create PR
2) Browser - interactive edit

(or) git fetch origin
git checkout -b new-heading origin/new-heading
git merge main

go to main, merge, push

1. git fetch origin

Just switch ← git checkout -b new-heading origin/new-heading

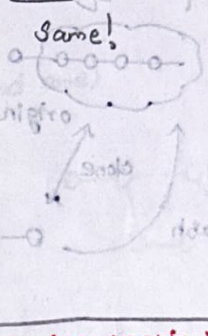
git merge main

why merge main: we can't touch main. 1. merge to branch, clean conflicks. Later merge with main (no conflicks)

2. git checkout main
git merge --no-ff new-heading
git push origin main

--no-ff → not fast forward (why? → No merge commit) - So to have merge commit.

(or)



1. git fetch origin
git switch my-new-feature
git merge master
→ fix conflicks

2. git switch master
git merge my-new-feature
git push origin master

Branch production rule (protection)

setting → Branches → Branch protection rules

* disable force pushing, prevent branches being delete - optionally status check before merging.

Branch name pattern : main

Require PR before merging: ✓

Note: Something updated in main: can't push

Requires: create branch & PR.

Open Source Contribution

clone & work after Forking!

* Fork & clone workflow: dev makes fork - push to their own (fork) repo. before PR

Common: large open source (1000s of contributors) only few maintainers.

Forking & Cloning workflow

* Manage access: can't be have many collaborators
* Fork: personal copy. (Github feature)

demo:

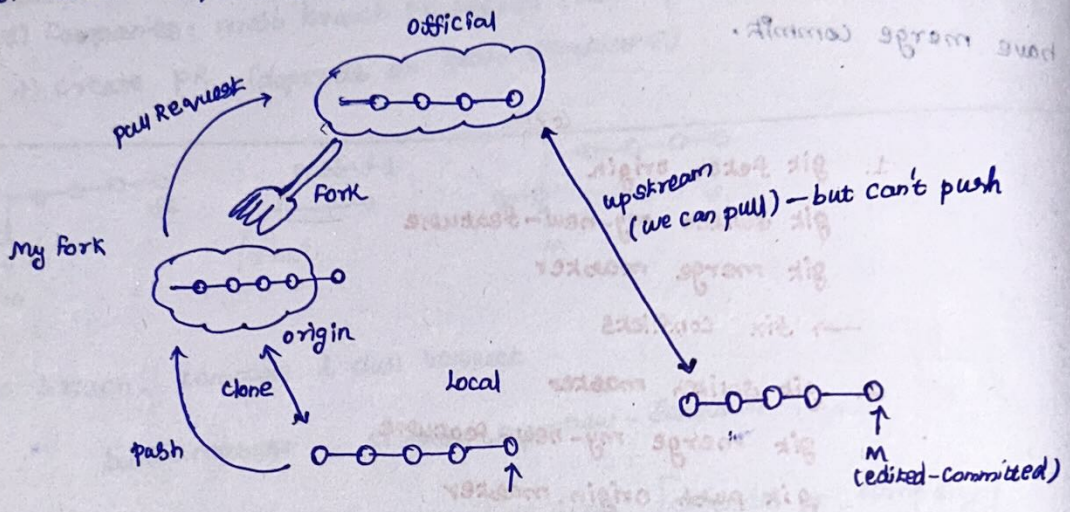
* Fork then clone! (push to own repo)

change origin to personal fork! (edit, commit, push)

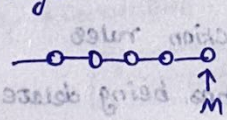
---

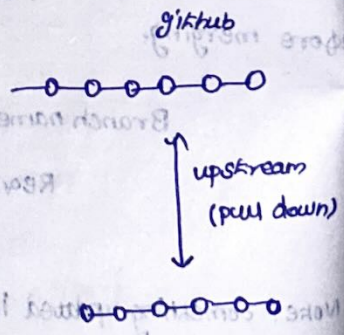
1 commit ahead: pull request (Share with original owner)

change: Create PR, Comments, Conversations, ... (Real contribution: that don't owned by me)



If they accept & Merge PR





New work in github:

1. pull down from original repo
2. push to My repo (forked)

---

git remote add upstream original url

↓

Common

pull

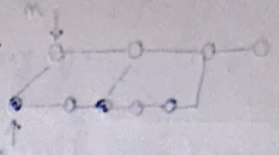
git pull upstream main (To take pull from org)

push

git push origin main

Rebasing - Scary (why?)

1. Try to Avoid rebasing at all : Some Companys.
2. useful – when not to use (must know).

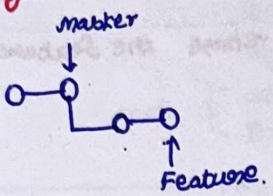


Rebasing

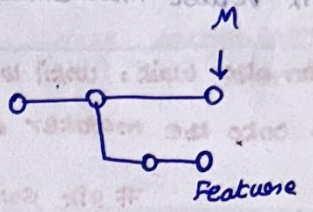
1. Alternative to merging
2. Cleanup tool

2 ways of Combining branches
1. merge 2. Rebase.

Rebasing:

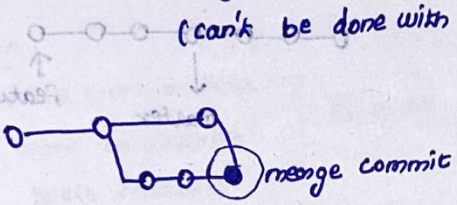


But before Some changes in main →

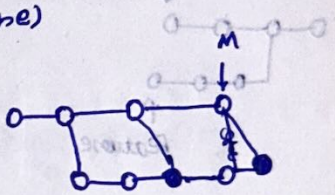


get new work from master to feature:

1. merge main to master branch.
(can't be done with fast forward one)



more work



Again a commit merge

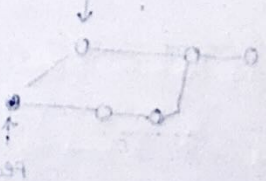
* Very active master branch: Continuous merging I need to do!
* It has lot of merge commits: unnecessary (not my work) – feature's history is muddled.

longer scale – Not Informative (since main has the history log of them)

* my branch's merging commit – don't add any value! (useless)

makes history: redundant, less clean

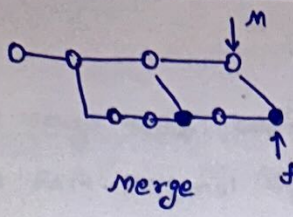
Rebasing to rescue



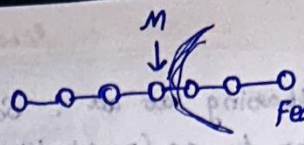
Rebasing: rewrite history

1. Rebase feature branch onto master – moves entire feature branch so that it begins at the tip of master branch.

Rewrite history – No merge Commit (Rebasing rewrites history by creating new commits for each of the org. feature branch commits)



Merge



Rebase (Rewrites history)

Rebase: we are coming with new base (Linear structure)

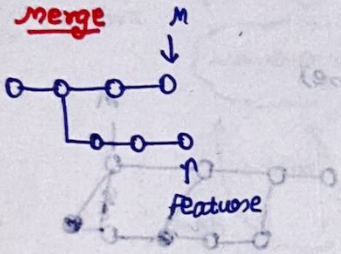
```
#git switch feature
#git rebase master.
```

---

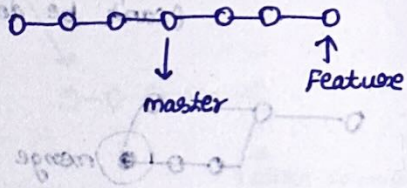
we can also wait: until we finished feature & then rebase the feature branch onto the master branch

```
#git switch feature
#git rebase master
```

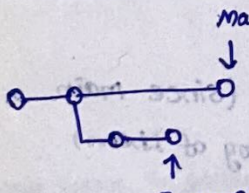
merge



Rebase



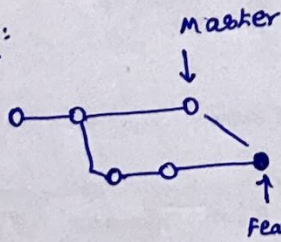
* huge no. of people (say 10) → merge commits ↑
* Rebase: Individual branch history!

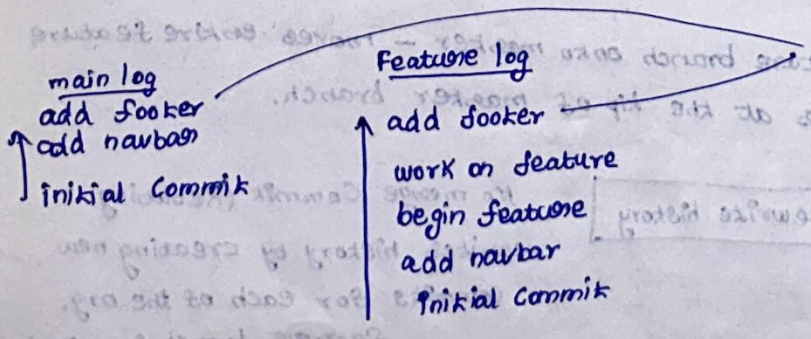


Demo

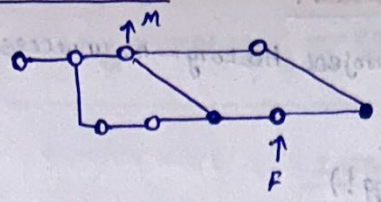
1. add website.txt — commit
2. edit website.txt — commit
3. Branch: feature.txt
   1. add file feature.txt, commit
   2. edit feature.txt → commit
4. switch: master : edit website.txt

merge:





more work on feature:



Again merging.

Now history of feature branch:

merge master into feat
main & mergeCommit. — add login form
more work on feature
mergeCommit ← merge 'master' in to feature
main ← add footer
work on feature
begin feature
main [add navbar
initial Commit

Note: merge commits there!

As master changes – merge commits ↑

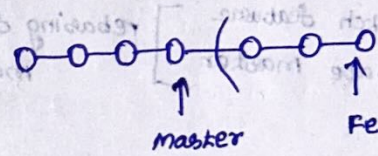
Rebasing:

git rebase master [In feature branch]

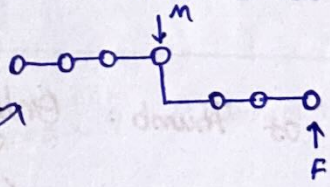
Feature
- more work on feat
- work on feature
- Begin feature

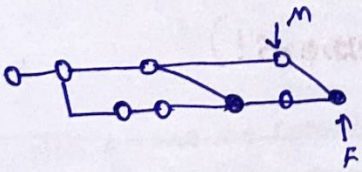
feature

main
- login form add
- add footer
- add navbar
- initial Commit



(or)



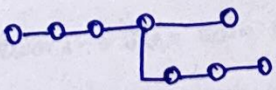


Rebasing
(re write history).

merged

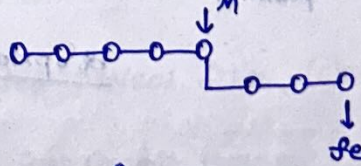
Adv: clear history tree
linear
Few commits (no merge commits)

Note: work still there, date there (meta data)
But : new commits! (not update) – completely
new commit – new history (cleaning history: make it easy to see)



After rebasing: a commit to master

git rebase master

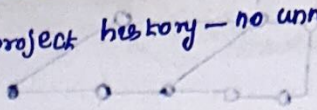


what we got?

git diff feat..master

Note: Begins (feature branch) at tip of master branch!

when not to rebase?



Why rebase: clean, linear structure (clean project history — no unnecessary commits, linear project history)

Rebase - rewrites history - (Warning!)

1. Never rebase commits that other people already have (shared)
2. If already pushed to remote:

   Don't rebase - unless (sure: no one - using those commits)

* Rewrite history — annoying to collaborators (different history)!
* Serious: You don't want to rewrite any git history that other people already have. pain to reconcile the alternate histories.

Rebase commits: you have (locally) & others don't!

Say: your own feature branch (you alone working)

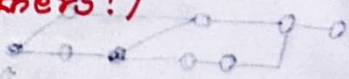
Don't rebase master branch.

```
git switch feature
git rebase master
```

rebasing feature not master branch!

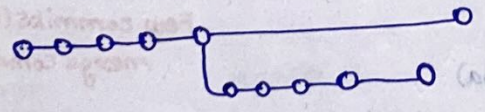
proper use of rebase (must) — else problem!

Rule of Thumb: Only Rebase - your own own work (not shared with others!)



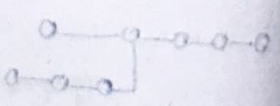
Conflict

website.txt (changed) - both main & feature.



```
# git rebase master
└→ Conflict → See gitKraken
```

'partial rebase' - ongoing



options:

* abort rebasing
* Rectify Conflicts.

```
# git rebase --abort
```

Solve conflicks

git rebase --continue

Interactive rebasing – clean up history

Rewrite, delete, Rename / Reorder commits (Before sharing) : Rebase

-i (interactive mode) ⟶ Specify how far we want to rewrite.

Note: Rebasing branch – (current) HEAD (Not other branches)

git rebase -i HEAD~4

git commits

2029e20

⋮

9 commits

Inikial

3b23c17 – Readme
2029e20 – Cat made
655204d – fix an nav
2a45e71 – fix navbar typo
4ff2290 – add top nav
6e39a76 – add SS
240827f – add html
5190ab6 – boilers html
cbee26b – projectfile
0e19c7a – init

goal

b8 58889 – top navbar
b4fa7d4 – bootstrap (SS replaced)
A273423 – base html
0ff b0aa – project file
0e19c7a – init

HEAD~9

Current branch

not recommended

git rebase -i HEAD~9

[current : main] feature branch

pick – use the commit
reword – use but edit commit message
edit – use commit, stop for amending (but)
fixup – use Commit contents – meld into prev commit & discard commit message
drop – Remove commit.

Rebasing : lists.

Reword – (Oldest ⟶ Newest)

pick → Keep it don't change anything
reword → Just edit message

In editor

reword cbee26b I added project files (closed file)

Now: opened that Commit message

Add project files (close)

now: # git log ⟶ 71boefb : Add project files

* Rewording - changes Commit hash (All others commits also)

we are changing only 1 commit message — All commit hashes changed in the taken one

10 commits (total) → taken 9 → only init hash remains same (since not taken) - all others change

HEAD ~ 9

why! hash generated: (lightly based on): previous hash! parent changes: child changes!

Combine Commits (redundant ones)

oops forgot JS] same!
add bootstrap]

Get rid of and (future) commit message combine to prev one!

* Squash: use commit, melt in to prev commit (doesn't discard commit message)
* fixup: like 'Squash' — discard this commit's log message.

(keep contents, delete commit message of and one)

pick 21acd17 add bootstrap
fixup 4C6Cb28 whoops forgot JS

Log:

3bod8ab: add bootstrap only

Removed

'Subsequent hash: Changed'

Remove: (Not content)

fix another navbar] contents
fix navbar typo ↓
add topnavbar ✓ (keep)

pick add topnavbar]
fixup navbar typo] contents goes to add topnavbar.
fixup another navbar]

Remove commit + changes

→ accidently: don't want that commit
→ say: app.JS committed no need

Created in other Commit(so remains) (app.JS Contents gone)

drop mycat done this → removed along with Content

Edit last commit message

```
# git commit --amend        (easy way)
```

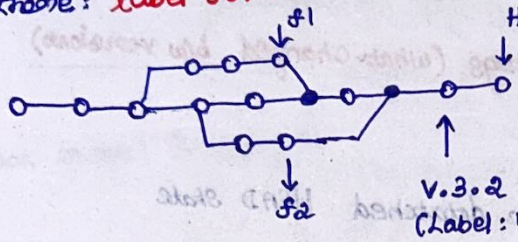
we can add stuff:
```
# git add .
# git commit --amend
```
] does the job.

Git tags (history)

Idea behind tags:

* pointers: refers to particular points in git history
* use: mark version releases (V4.1.0, V4.1.4 etc..)
* Branch references don't change (once created – always

there: label for that commit)



f_1, f_2, HEAD → changes!

Tag: V1.0
V2.0

List tags:

Specific ones (wildcard)
git tag -l "V1.8.5*"

```
git tag
V1.0
V2.0
```

Tag: pointer (refers a commit)

Types → Light weight tags (name / label to commit)
→ Annotated tag (extra data (meta): author name, email, date, tag message)

Semantic Versioning:

Standard for software releases: 2.4.1 (seperated by periods)

meaning to releases.

1. Major version – Incompatible API changes
2. minor – add functionality in backwards compatible manner
3. patch – fix bugs

Major. minor. patch

2. 1. 4

semver.org

0.y.z (initial development)

Once version released: Contents shouldn't be modified.

1.0.0-alpha (pre-release)
1.0.0-beta

* Initial release: 1.0.0
  - ↳ patch (no features / no major changes / breaking changes)
  - ↳ Functionality (backward compatible) – optional (Shouldn't force)
  - ↳ not backward comp – Removed / modified features

View & Search tags

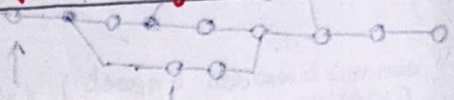
facebook/react

> git tag [print all tags in the current repo]

Search: > git tag -l "*beta*"

↳ list

Compare tags (what changed b/w versions)



> git checkout 15.3.1

we can create branches from the detatched HEAD state

> git switch -c BRANCH_FROM_TAG

> git diff V17.0.0..V17.0.1

Creating light weight tags

> git tag <tag-name> [default: HEAD reference taken]

Annotated tag

git tag -a <tag-name>

-m. → pass message (inline): else open in editor!

> git show V17.1.0 tag gen, date, message

Tagging previous commits

git tag <tagname> <Commit> (hash)

git tag -a V17.2.1 b7687e5a

Forcing tags (rare)

*we can't reuse a tag (already referring to a commit)

-f → force own tags.

git tag -f <tagname>

> git tag v17.0.3 696e736be

fatal: tag 'v17.0.3' already exits

Force that tag move here (I made a mistake) ← > git tag -f v17.0.3 696e736be

Deleting tags

> git tag -d <tagname>

Pushing tags

1. By default

git push → won't pushes tags to remote repo

git push --tags (to push all tags) – already not there! (new tags)

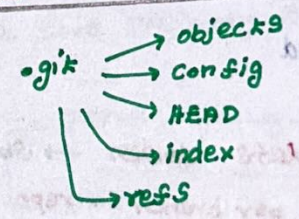
git push origin v1.5 (single tag)

---

github: tag menu (See tags)!

Git – Behind the scenes (hash & objects)

Local config files:



Config: for configuration (global settings like global username, email across all git repo (per-repobasis) → only for a particular repo.

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = https...
    fetch = +refs/heads/*:refs/remote/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

[remote "colt"]

url =

fetch = +refs/heads/* : refs/remotes/colt/*

See documentation:

git config user.name ⟶ global

git config --local user.name "local name"

In config file:

[user]

user = local name

Play with Colors: (using config):

google: git config color

[color]

ui = true

[color "branch"] ⟶ how remote branch look like

local = cyan bold ⟶ see doc for more colors

current = yellow bold

[color "diff"]

old = magenta bold

---

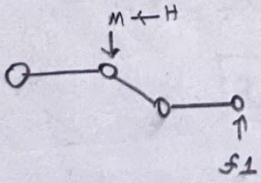
refs folder ⟶ Stores pointers (tags, head)

refs/heads : 1 file per branch in repo

each file named after branch & has hash commit at the tip of branch

refs/heads/master ⟶ Commit hash of master branch (latest commit)

refs/tags ⟶ one file for every tag!



each commit: updates ref file of that branch

heads | Remote | ..... tags .....

origin

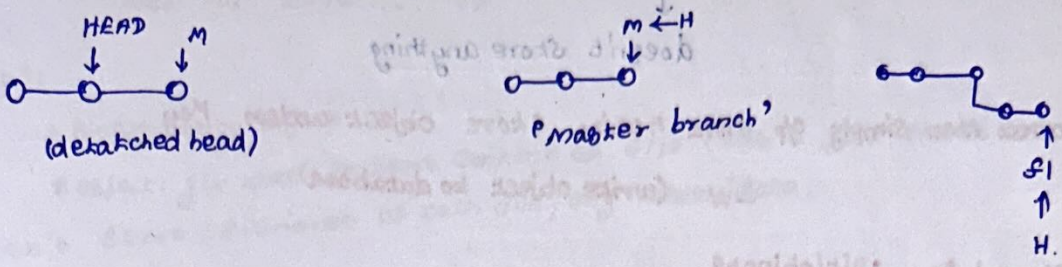
colt

origin ⟶ HEAD ⟶ Keep track of origin master.

heads ⟶ ref of branches.

HEAD File

* txt file: keep track of current branch (hash)

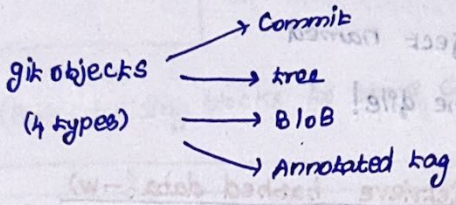


Objects folder

1. core of git - Commits, Contents live here (Contents, Backups)
2. Compressed, encrypted.

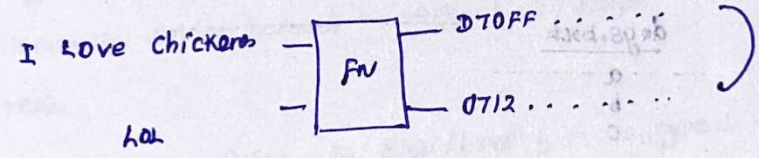
2 digit: hex decimal folder names

* git stores: full snapshots



Hashing: (hash codes): hexadecimal ones

hashing function: map i/p to some arbitrary size to fixed-size o/p values



Cryptographic hash functions:

1. Infeasible to invert: One way function
2. Small change in i/p: large o/p change
3. Same i/p - same o/p
4. Unlikely: find 2 i/p with same o/p.

eg: SHA1 (40 digit: hex decimal numbers) - GIT

Git Database

* key: value datastore (GIT) - insert any kind of content → unique key
* Later Retrieve content

keys: Checksum of SHA1.

>echo 'hello' | git hash-object --stdin [hash]

pipe to git

--stdin → stdin rather than file.

Take content → unique key generate (value: content) stored. (used to store content)

But : `>git hash-object <file>` creates hash

↓

doesn't store anything

Rather than simply o/p SHA1 hash, store object under key

`-w` (write object to database)

Stored in `.git/objects`

```
echo "hello" | git hash-object --stdin -w
```

hello → ce..... 40digit blob

folder name : ce (1st 2 digits)

file name : 01362... (encrypted, compressed) - binary - Last 38 digits to name file.

1st 2 dig → folder in object named

next 38 dig → naming the file!

Retrieve hashed data (-w)

`>git cat-file -p <object-hash>`

Note :

dogs.txt
a
b

dogs.txt
a
b
c

hash & store use `-w`

Store after hashing

`git cat-file fd915 -p`

a
b
c

`git cat-file -p 39e276l`

~~fail~~

b

Eventhough edited : we have both versions available.

As long as deleting repo.

*we have both versions! (retrievable)

Restore:

`git cat-file -p fd9150c2a > dogs.txt`

(prev) version.

So git stores all the hashed & stored -w objects!

* Even file is edited – hash-stored objects won't be changed by git

'hello' → 1f7a 3a

Blobs:

* Binary large object

* object: git uses to store content of file (blob – just stores content) – doesn't store : filenames of each file / any other data.

1f7a7a47... → Name of file (1st 2dig: folder name).

blob

```
//main game code
Console.log("hello world");
//more code
for(let i=0; i<10; i++){
    Console.log("oink");
}
```

→ Content stored in comp & encrypted mode.

git → keeps track of filenames (trees)

100644	test
040000	images
100644	index.html
100644	main.js

c3871 9da....

blob has hash – (Blob : building blocks to store content)

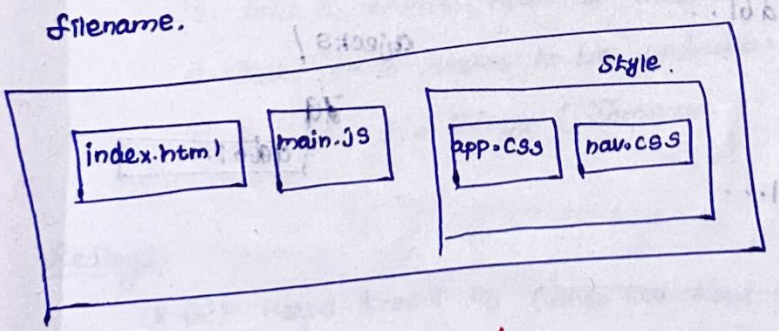
Trees

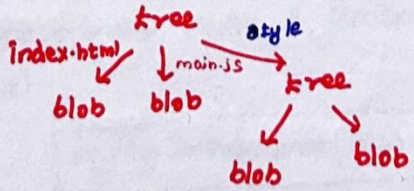
* entries & references to blob

* git object to store contents of a directory (filenames, folder names) – refers blob / other trees.

* Each entry – SHA1 of blob / tree + mode, type & filename.

tree		
blob	1f7a	app.js
tree	9828	img
blob	be32	README.md
tree	8aff	styles
type	hash	filename





pointers to blobs & trees

1) 1 folder 1 file
So 1 tree with a blob ref

2) 1 folder, 1 file, 1 subf
1 tree with a blob & tree.

* each tree & blob has hash keys (reference)

*

View trees

>git cat-file -p master^{tree}

git cat-file → prints git objects

master^{tree} → tree object pointed by the tip of master branch.

git cat-file -p master^{tree}
↳ pretty print

git cat-file -p tree-hash
↳ 1ea0baa

Type of file: # git cat-file -t 5C407....
tree (o/p)

tree : file managent entries to blobs & tree + hash, type

Commits

fa49b07...

Commit	
tree	C387...
parent	ae23...
author	Sirius
Committer	Sirius
message: this is my commit.	

* Commit: Combine a tree object along with info about context that led to current tree.

tree : Current Content of staging area.

98 7fa ← 23bff6 ← master

The parent of 23bff6 is 987fa
↓
Current HEAD
then change HEAD to 23bff6.

create, edit - dogs.txt, Commit (take hash)
git cat-file -t 79dd61...
↳ type

objects/
79
dd61....

Commit
git cat-file -p 79dd61...
tree 13205...
author
Committer
message

git cat-file -p 13205... → tree
has one blob : dogs.txt (5d9150...
Create cats.txt, edit, ... Commit
git cat-file -p 5119...
tree a6de9...
Parent Commit : 79dd61.

inspect tree: git cat-file -p abde9c..

2 blobs — cats.txt bf6544
dogs.txt dd9150c2

Note: we don't changed dogs.txt (So can't changed dogs.txt)
See same blob hash

Truth: It is the same blob reference!

1) So no change means: Keep the hash (no need to store again)
2) If any change: again hash – store (Store the reference in tree)!

Deeper

1. Each Commit: Snapshot (changed: hash-store take reference
No change: Keep hash) is stored in tree (Commit's tree)

when we jump to a commit: headless state
Just go to snapshots & decrypt blobs → show that files'!
In a tree of that commit

when dogs.txt is edited, committed
1. The blob hash for it is changed for good.

→ tag object: same – each tag – stored reference after hashing

---

Reflogs (reset)

1. Lost a commit / rebased (when shouldn't)
2. Undo (when seems to be undoable)
3. Reflogs – fix things (Come out of bad situations)

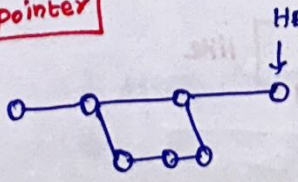
Reflogs:

* Git keeps track of (when the tips of branches & other ref were updated in the repo – (view those & update those reflogs – using git reflog command)

reflog – Reference logs → Just logs (whenever references got updated)

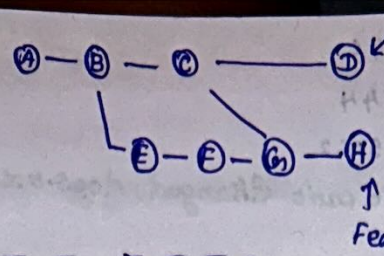
transcript / record: git keeps.

Reference / pointer



Different references

HEAD
MASTER
remote reference
fetch head
merge head



Reflog: (For HEAD)

- Commit G : New commit
- Commit H - New commit
- Commit D - Switched to main from feature
- Commit H - Switched to feature from main

- .git/logs/refs
  - HEAD (not compressed file)

» whenever HEAD changes — log entry is made

Limitation of reflogs

* Reflogs : local (won't be shared with collaborators)
* They expire : (once 90 days — cleaned) — But we can configure

git reflog (Show: commonly used) — Show specific log reference.

> git reflog delete
> git reflog expire

] not used by end users

git reflog show HEAD (reference) [default: HEAD]
git reflog show main [logs for tip of main branch]
git reflog

How git reflog show different?

git logs → How?

1. Includes more info: whether checkedout of a branch
   git log (Just have: Commit Entries)
2. History of things: even after rebasings — log remains same!

rebased: git log (hash gone) but in reflog (log) there!

Use: Reference reflog entries to other commands

each line has its own reflog identifier.

HEAD@{13}
{12}
:

] like

git reflog show donkey

Only 2 logs:

1. when Created that branch
2. when made a commit

Reflog references

`# name@{qualifier}` – entry of a ref log

1. Access specific git refs : name@{qualifier}
2. use this syntax to access a specific ref pointer (pass to other Commands – checkout, reset, merge)

HEAD@{2} ⟶ where HEAD was 2 commits/moves ago.

git reflog show HEAD@{10}

(from HEAD@{10} to HEAD@{0}) [From early beginning of repo: own case: HEAD@{57} to HEAD@{10})

git checkout HEAD~2 ⟶ 2 commits before (Same branch)

git checkout HEAD@{2} ⟶ log (previous) – may be in some other branch – go there (or detatched head)

git diff HEAD@{0} HEAD@{5}

Both : what changed in the branch!

Time-based Reflog references (Rescue time)!

* Every single reflog entry has time stamps!

Ref : timestamps

> git reflog master@{one.week.ago}
> git checkout bugfix@{2.days.ago}
> git diff main@{0} main@{yesterday}
> git reflog show HEAD@{one.month.ago}
> git checkout master@{1.week.ago}

Reflogs Rescue – hard reset

1. Access commits – seem lost & are not appearing in git log.

Logs : repo

create veggies.txt, commit
edit, commit (veggies)
edit, commit (more green)
edit, commit (Summer veggies)

Now: get rid of this commit

→ > git reset --hard db727ba [lose work & log altered]

Lost

I want those changes back

> git reflog show master

take hash : fb5072a [master@{1}]

→ > git reset --hard master@{1}

Restored!

Restored!

Note: 1. reflog - local changes

2. git won't delete blob, commit, tree objects when rebased/ reset / anything (where commit hash unavailable)

3. Anytime when hash is found (reflog)

1. restore to that point

Why? : All commits have trees having snapshot of time!

Undo a Rebase

1. Rebase — rewrites history

Create branch flowers, flowers.txt, commit
add flowers, commit
more flowers, commit

5 Commits in log for flowers.

git rebase -i HEAD~4 [1 to 4 Commits log]

pick abbodic add flowers
fixup 14fa.. add jasmine
fixup 355885f add lilly
fixup c47ecfc add rose

4 Commit → 1 Commit + 1 = 2 Commits in flowers

reword
fixup

Now: only 1 Commit

Now I want to get rid of rebasing

git reflog show flowers

get hash: flowers{2} (before rebasing)

>git reset --hard c47ecfc

>git log

Restored!

every commit has its parent!

Aliases

1. Define & put them!

Global config file:

~/.gitconfig
~/.config/git/config
] global - systemwide!

write aliases

*alias: Own command for some git commands (user experience)

git ci → for → git commit

git lg → for → git log

[alias]
s = status
l = log

[alias]
s = status
l = log

From Command line

git config --global alias.showmebranches branch

Aliases with Arguments

git alias: appends it to end

cm = commit -m

means

git cm "my message" ⟹ git commit -m "my message"

[alias]
cm = commit -m
a = add

Useful aliases

→ Reddit, blogposts

→ @durdn : must have git aliases : adv examples [formatted way]

→ gist.github.com/mwhite/6887990

→ github : GitAlias/gitalias