Github

Introducing Forking

we need contributers and maintainers.
There can be 1000 of people contributing but only handful of maintainers and owners of the project.
so something to manage this.

Fork & Clone: Another Workflow

The "fork & clone" workflow is different from anything we've seen so far. Instead of just one centralized Github repository, every developer has their own Github repository in addition to the "main" repo. Developers make changes and push to their own forks before making pull requests.

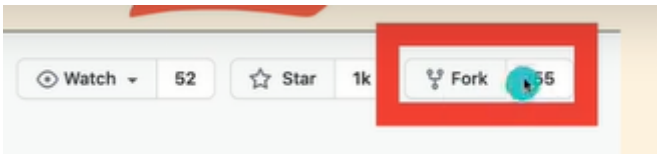
It's very commonly used on large open-source projects where there may be thousands of contributors with only a couple maintainers.

Forking

Github (and similar tools) allow us to create personal copies of other peoples' repositories. We call those copies a "fork" of the original.

When we fork a repo, we're basically asking Github "Make me my own copy of this repo please"

As with pull requests, forking is not a Git feature. The ability to fork is implemented by Github.



owner/repo_name

The original repo...

aapatre / Automatic-Udemy-Course-Enroller-GET-PAID-UDEMY-COURSES-for-FREE

My newly-created fork...

Colt / Automatic-Udemy-Course-Enroller-GET-PAID-UDEMY-COURSES-for-FREE

forked from aapatre/Automatic-Udemy-Course-Enroller-GET-PAID-UDEMY-COURSES-for-FREE

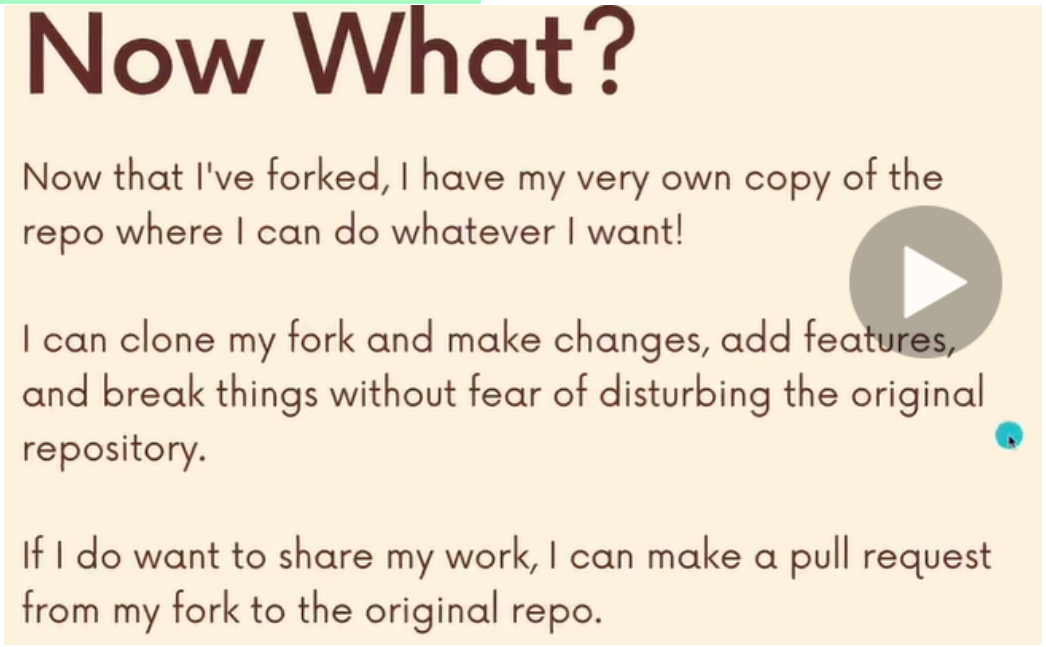
kazim/repo_name

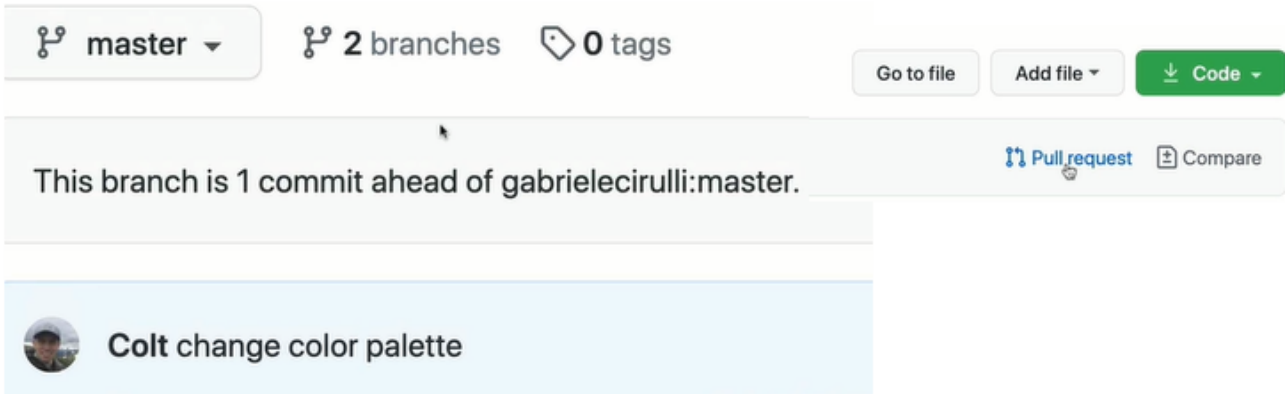
Forking Demonstration

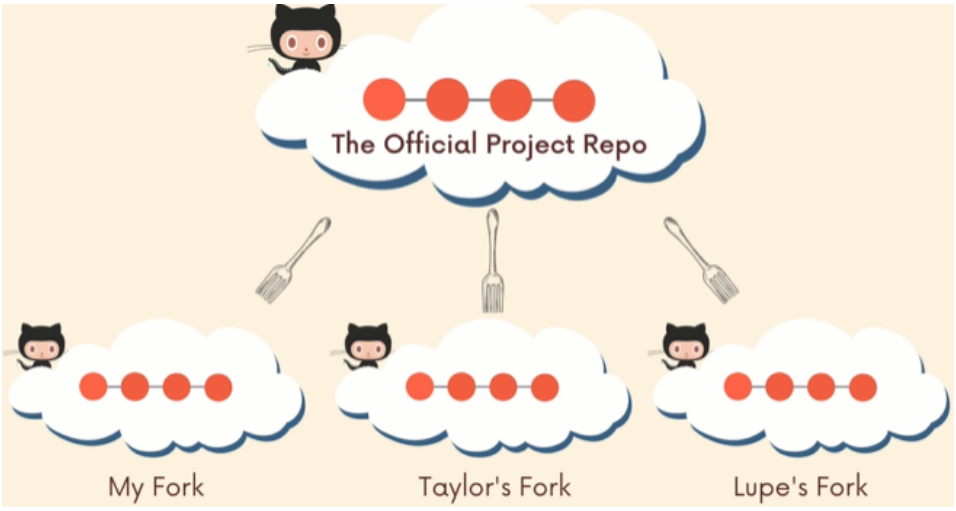
After forking, You can use the repos as your. Add collaborators and work on it. Or Maybe Later make a pull request to the Owner, aka contributing to an Open Source project

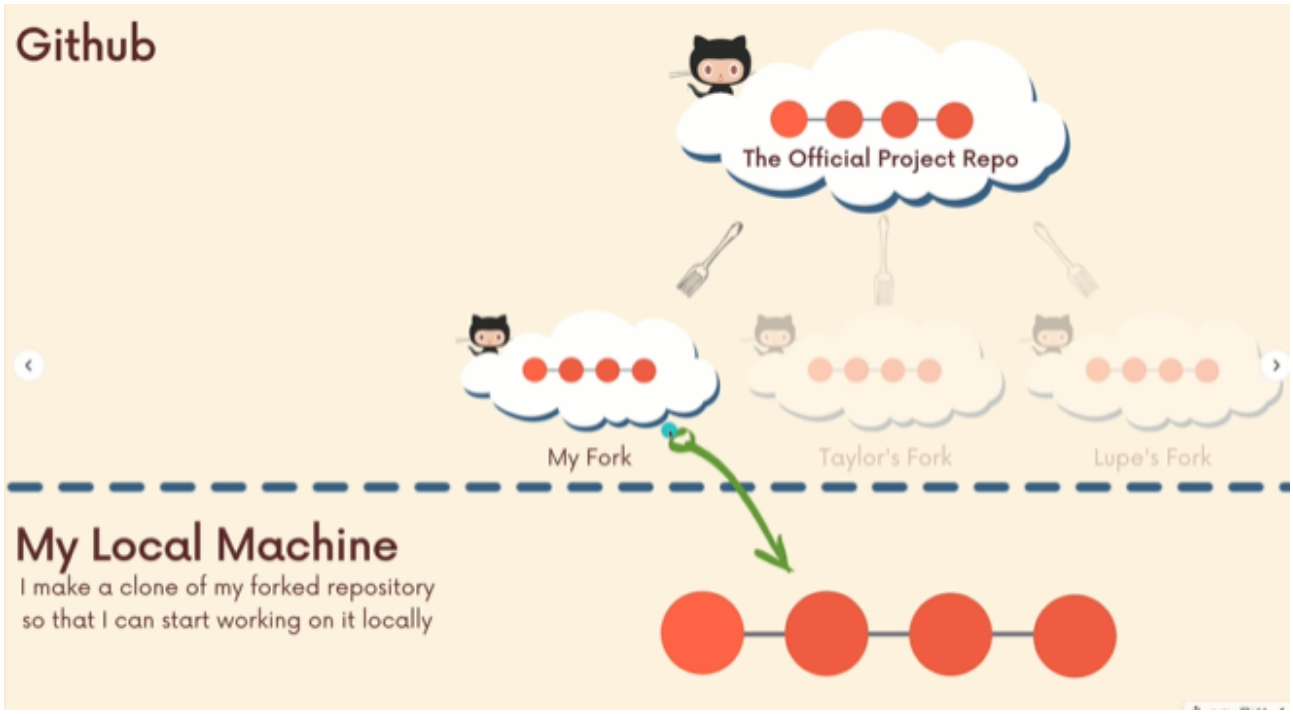
```
2048 ❯ git log --oneline
2048 ❯ git remote -v
origin  git@github.com:Colt/2048.git (fetch)
origin  git@github.com:Colt/2048.git (push)
```

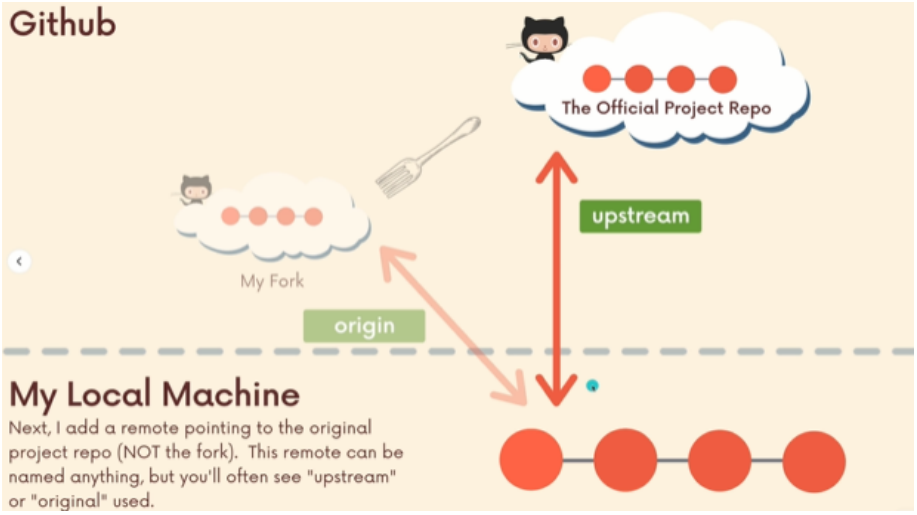
The Fork & Clone Workflow




master
2 branches
0 tags
Go to file
Add file
Code
This branch is 1 commit ahead of gabrielecirulli:master.
Pull request
Compare
Colt change color palette


The Official Project Repo
My Fork
Taylor's Fork
Lupe's Fork

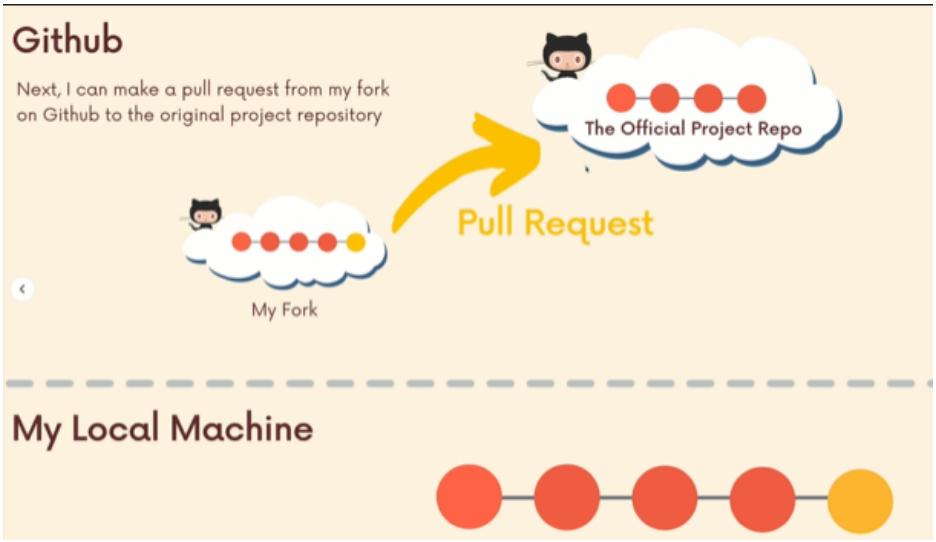

Github
The Official Project Repo
My Fork
Taylor's Fork
Lupe's Fork
My Local Machine
I make a clone of my forked repository
so that I can start working on it locally



Above, i have an automatic remote setup called origin, just because i clone this repository. Default remote name is origin. So origin refers to my fork. Then i am gonna setup a second remote, it would be anything i called, normally its called upstream. (Conventional name)

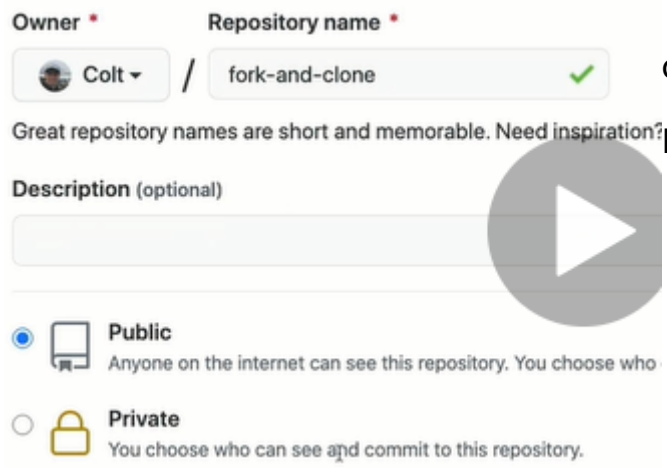
It refers to the original repo that we forked.

This allow me a place, where i can do my own changes and push changes into my fork. Also if there is any changes that have been made to the original repository on a large open source project called react. There might be new commits, pull requests, merge daily. I'll be able to get that down in my machine. So even if i cant push up i can still pull down.



1. FORK THE PROJECT
2. CLONE THE FORK
3. ADD UPSTREAM REMOTE
4. DO SOME WORK
5. PUSH TO ORIGIN
6. OPEN PR

Fork & Clone Workflow Demonstration

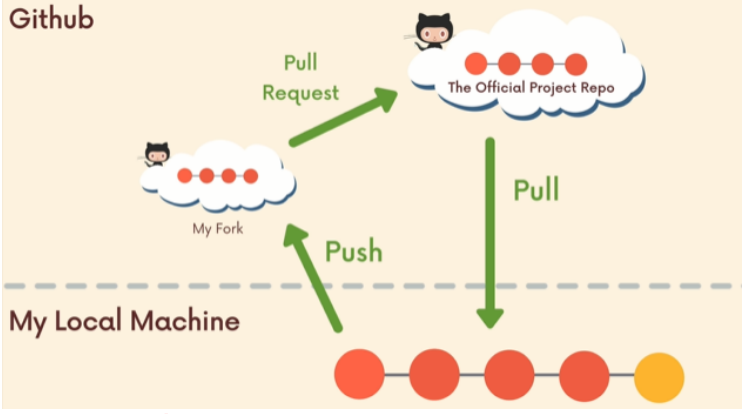


I am not gonna make anybody a collaborator.

Nobody will be able to push to this.

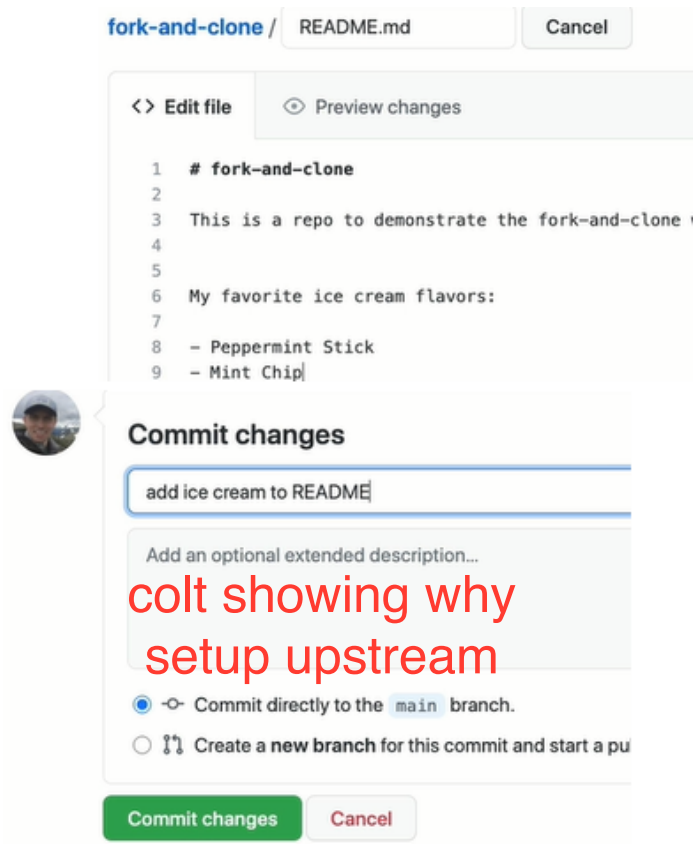
```
Stevie ❯ git clone git@github.com:StevieChic
s/fork-and-clone.git
```

```
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(point where the river/road split into two). fork and then clone

```
fork-and-clone ❯ git remote add upstream https://github.com/Co
t/fork-and-clone.git
```



colt showing why setup upstream

We(stevie) fork and clone the colt repo. we have the remote setup by itself. master --> origin/master. Basically FC, creates a copy on our github. Case: There can be new push on the original colt repo. so we might wanna setup the upstream(destination to colt repo). we cant push on colt repo but we can pull.
we can do whatever the hell we want on our github.

```
fork-and-clone ❯ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

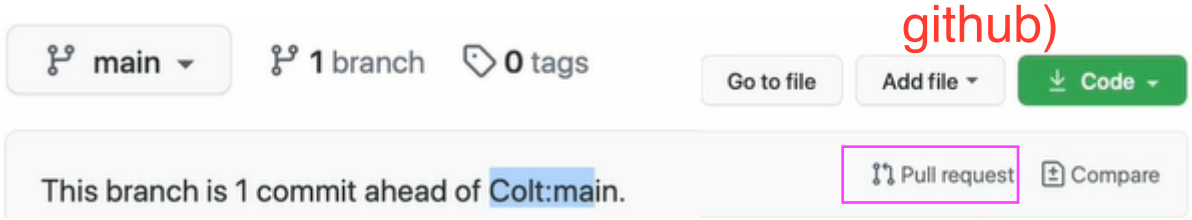
```
fork-and-clone ❯ git pull upstream main
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/Colt/fork-and-clone
 * branch            main       -> FETCH_HEAD
 * [new branch]      main       -> upstream/main
Updating 698b6ca..3c27aea
Fast-forward
 README.md | 6 ++++++
 1 file changed, 6 insertions(+)
```

```
fork-and-clone ❯ git add README.md                main
fork-and-clone ❯ git commit -m "add stevie fav ice cream flavors"
```

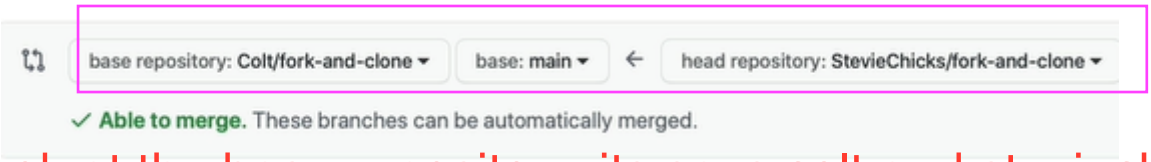
```
fork-and-clone ❯ git remote -v                    main
origin  git@github.com:StevieChicks/fork-and-clone.git (fetch)
origin  git@github.com:StevieChicks/fork-and-clone.git (push)
upstream        https://github.com/Colt/fork-and-clone.git (fetch)
upstream        https://github.com/Colt/fork-and-clone.git (push)
```

```
fork-and-clone ❯ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
```

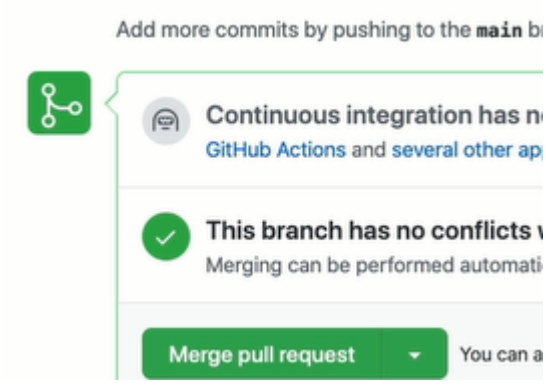
here, we are pushing to our copy(clone on our github)



It is recommended that if u are starting ur day, you might wanna pull first and then create a branch and work on it.



look at the base repository. its says colt and steviechicks



here, colt sees stevie merge request and he is happy with it and he merges that in

Use git pull to stay up to date with others work.

Rebasing : The Git Scariest Command

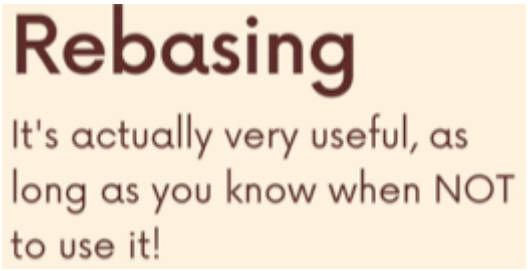
When colt first learn git he was told to avoid rebasing if possible.

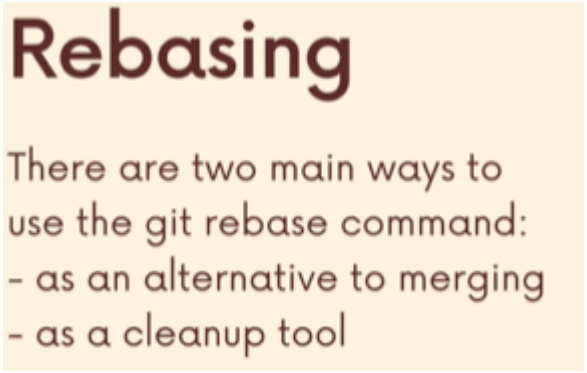
Rebasing is part of workflows or companies workflows on engineering teams.

Rebasing and merging, two ways of integrating changes from different branches.

Why is Rebasing Scary? Is it?

There's a divide over the git community.

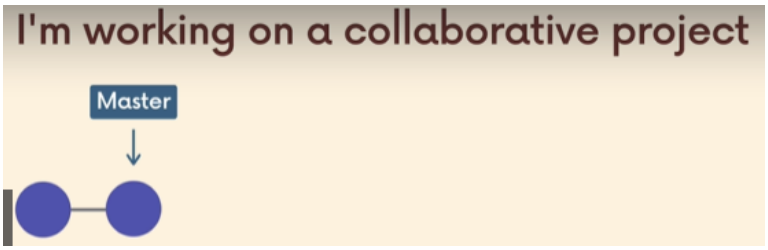


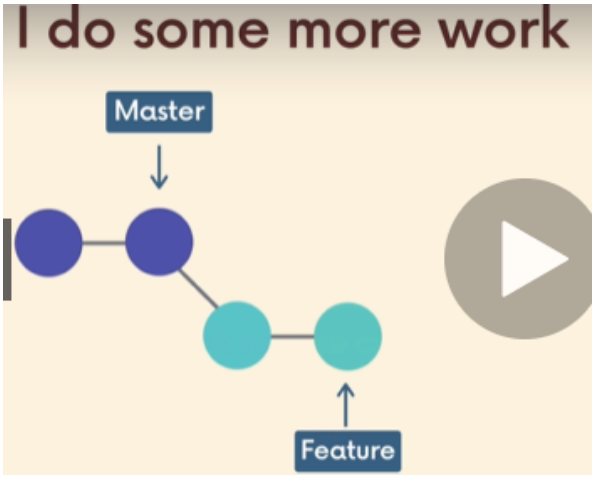


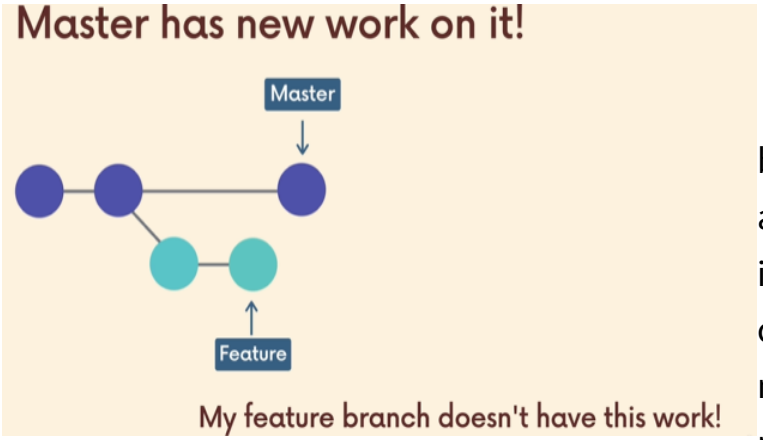
There is a divide in the community, cuz rebasing is an alternative to merging.
Some companies will ask their developers to use git rebase instead of merging. But other people get by just fine with git merge and they avoid get rebase.
So they are two big workflows for combining branches.
One is merging and other is rebase.

Comparing Merging & Rebasing

Lets discuss about the problem with merging or something Rebasing address that merging does not.



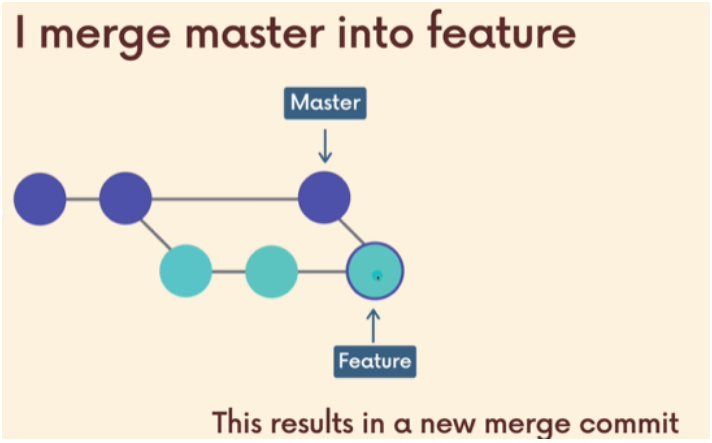




Lets say a fellow coworker did a bug fix and committed and the audit review merge the code, while i am still working and i have made couple of commits(checkpoints) in my working directory, i havent push anything or made a pull request.

I proabably want that new work in my feature branch. Maybe there's a bug fix or something. I dont wanna diverge from master branch for super long time without getting those new changes.

So how do i get that work from the master to my feature branch.

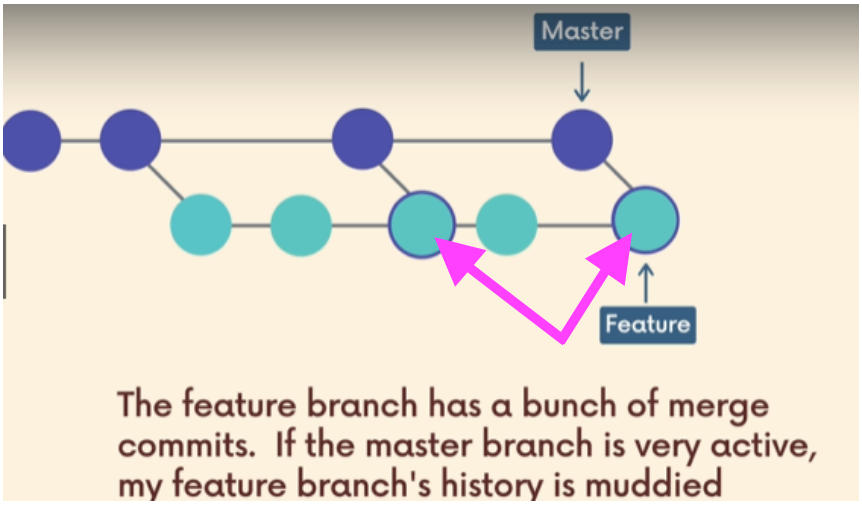


The next day you wake and see there's a new commit by ur coworker.

The late night commit from u on ur local repo

after we are sync with master branch, we continue our work. We see the next day, there is more work from coworker, our feature is behind of master by 'x' commits.

As i continue to work, so instead of 5 commits on my feature branch i have whole bunch of commits on my feature branch. Maybe i am working over the course of weaks. The rest of the team is very active. Lots of changes happening at the meantime.



So my entire feature branch has dozen of merge commits, that dont say anything about what i am working on, they have nothing to do with my actual code or commit. They just have to be there bcoz i merge.

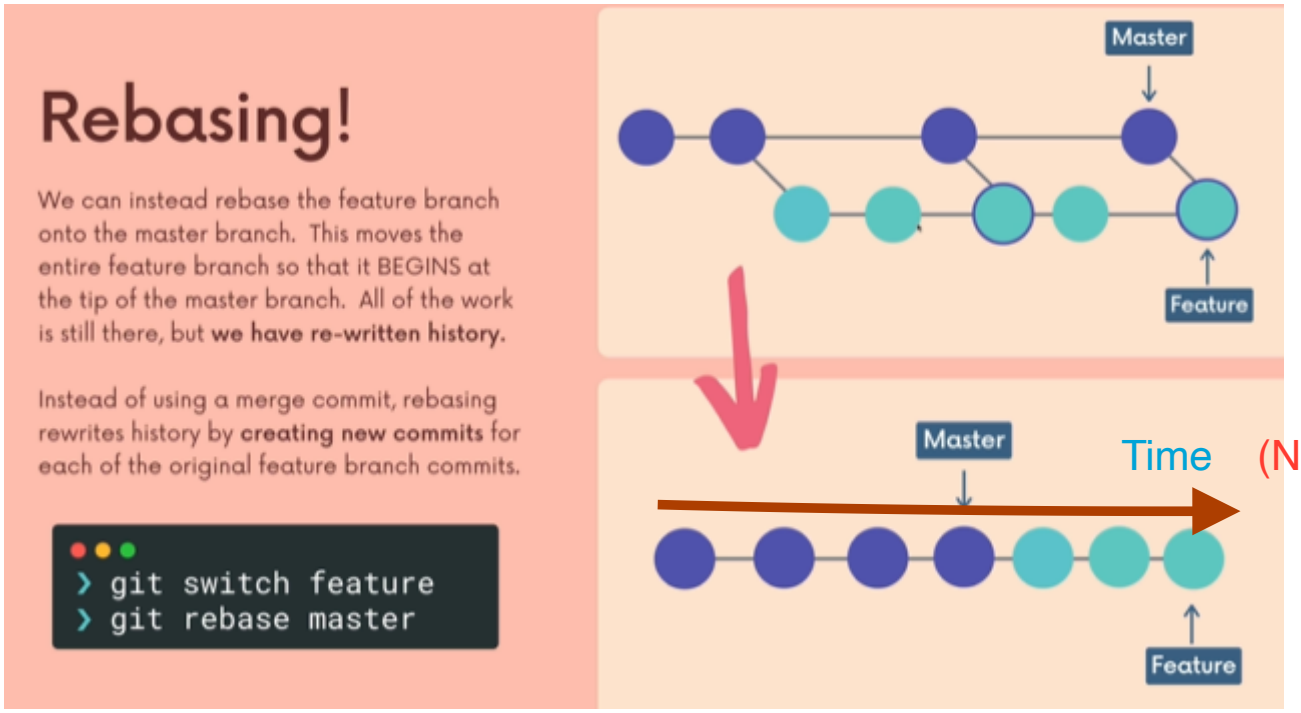
Just like me, its happening to others too.

If everyone else has merge changes into their origin/master. They need to get those changes into their feature branches.

So everyone feature branches might be riddled with bunch of merge commits.

When after fixing when they merge into origin/master, there is just bunch of useless merge commits as part of the history.

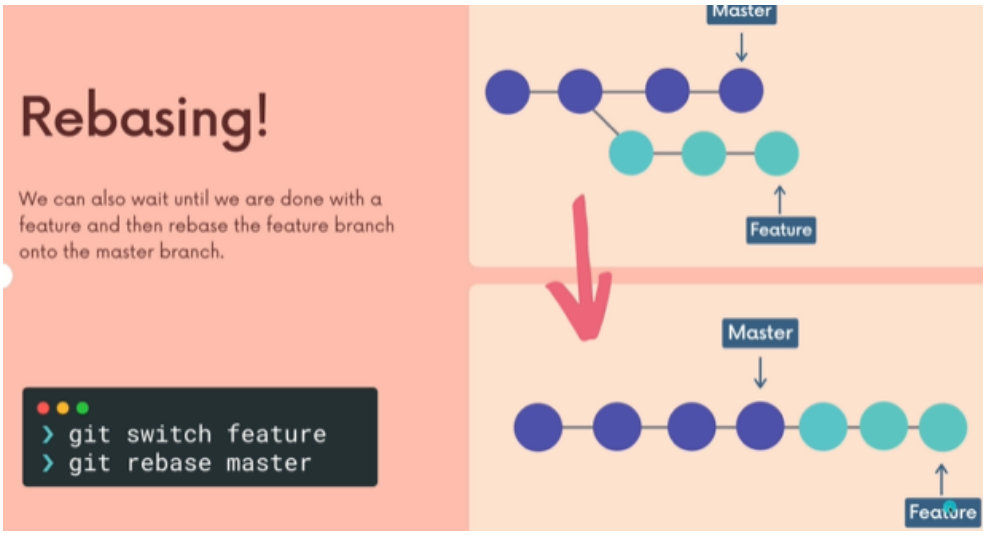
Not useless, but not informative. That makes our history muddier.



We still have the 3 commits on feature branch and 4 commits on master branch.

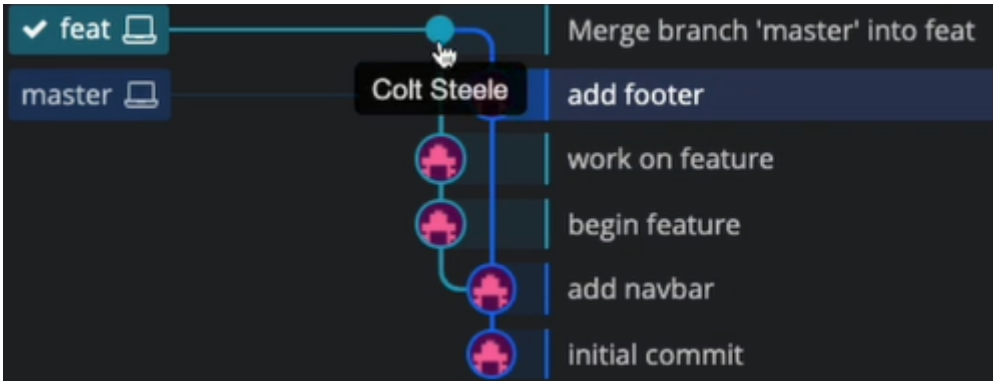
It has been re-written, it doesnt indicate the time when things were made. Each commit has data so that is preserved. The commits are moved around in order to give us this new structure.

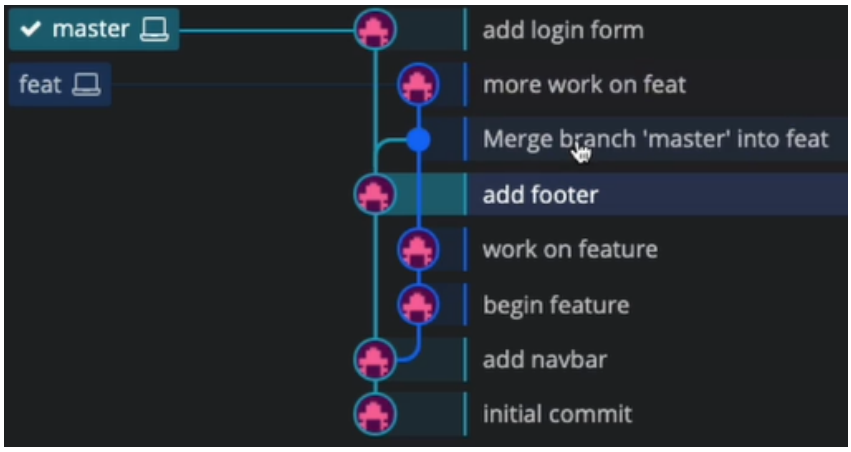
Date

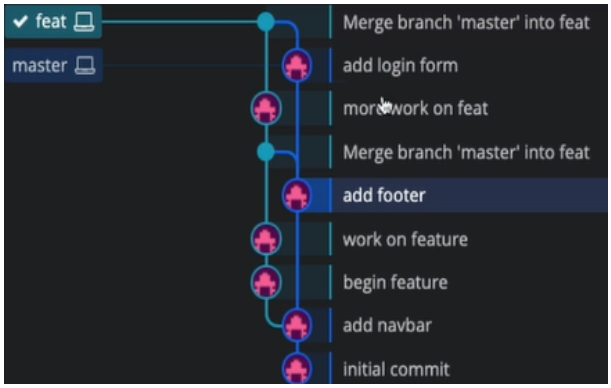


Its a linear,focused structure, easily readable that these are the commits i have done with my branch and the changes i made vs having merge commits * 10 ppl working on it just like us.

Rebase Demp Pt 1: Setup & Merging

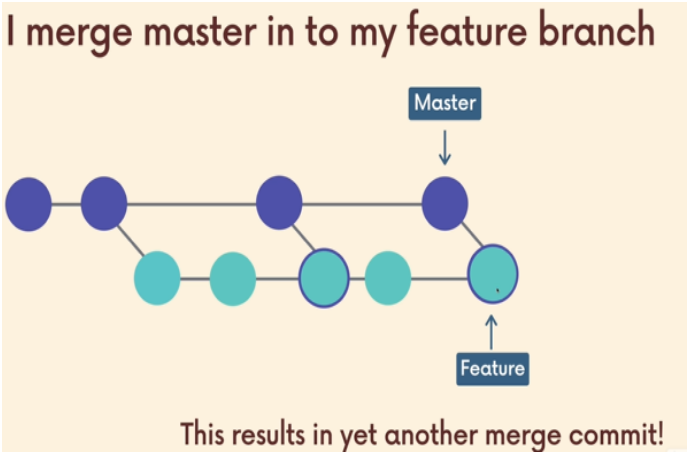






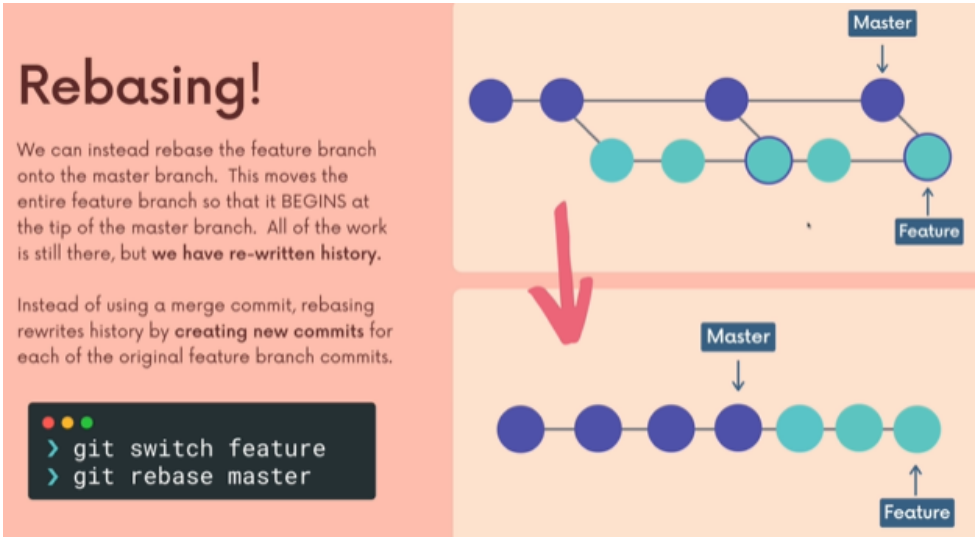
```
8b93760 (HEAD -> feat) Merge branch 'master' into fe
at
c3ec574 (master) add login form
33d06bc more work on feat
d9c49b6 Merge branch 'master' into feat
e19d409 add footer
c2b44e6 work on feature
8894594 begin feature
34f6bd0 add navbar
2be74d9 initial commit
(END)
```

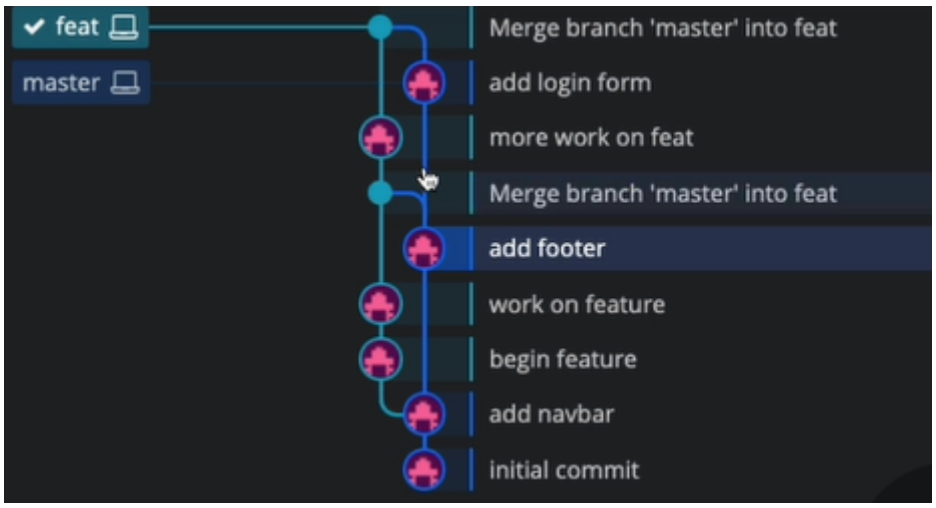
The commit history is harder to follow.



For a large project with just 100ppls with lots of commits, it would be way easier if we have a clean commit history. Where the commits are group together in a logical way and you minimize the no of those merge commits.

Rebasing Demo Pt2 : Actually Rebasing



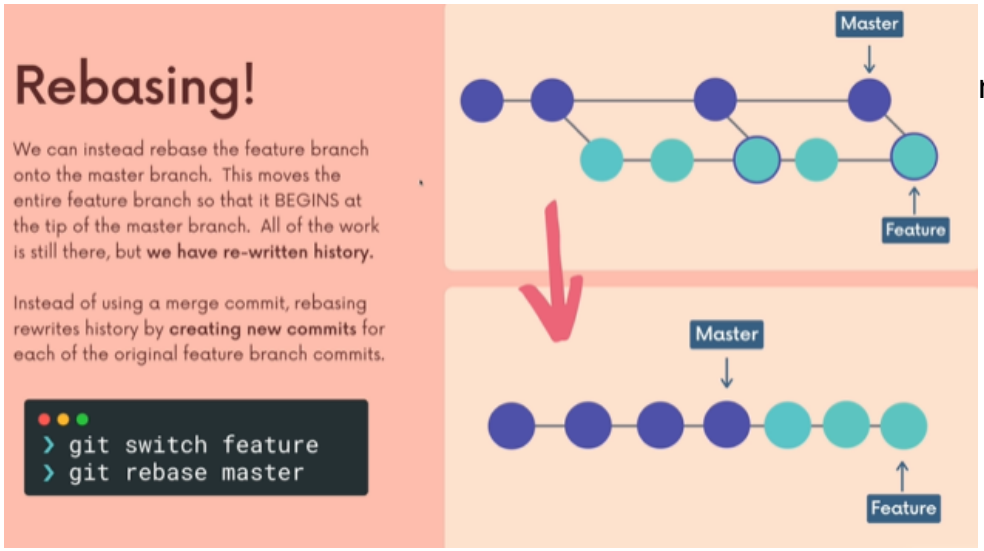


for Feat Branch, Above, i have twice merged in changes from master.

So the master branch doesnt have any of my feature stuff. I dont wanna merge anything directly into master rn. Instead, i am just concerned of getting the changes of my coworkers that are constantly adding to master.

Pull request are merged in. That gives me merge commits.

So, instead i am going to rebase. When we use the git rebase command. We are rewriting the history.



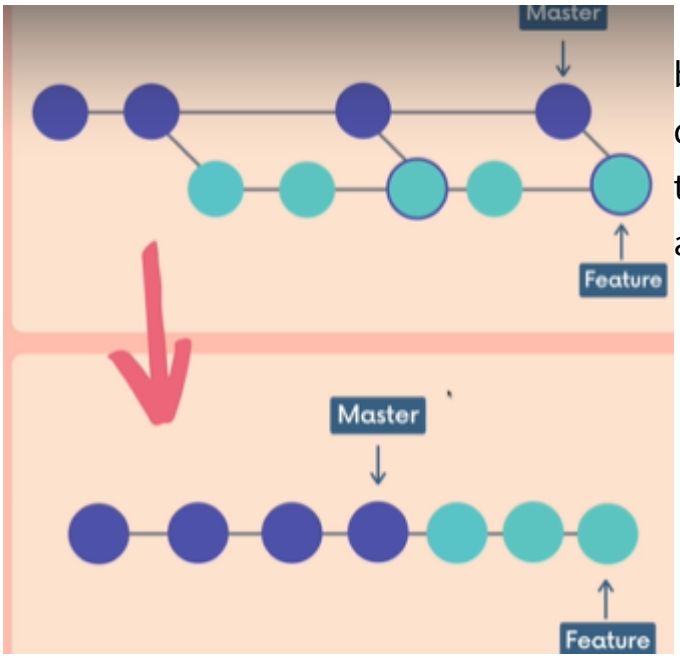
The merges are removed

Above is the actual history, when we use rebase the history get switch around a bit. Infact we get new commits generated.

Instead of writing merge commits for us, when we use git rebase, it rewrites history and creates brand new history for each of the original feature branch commits.

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

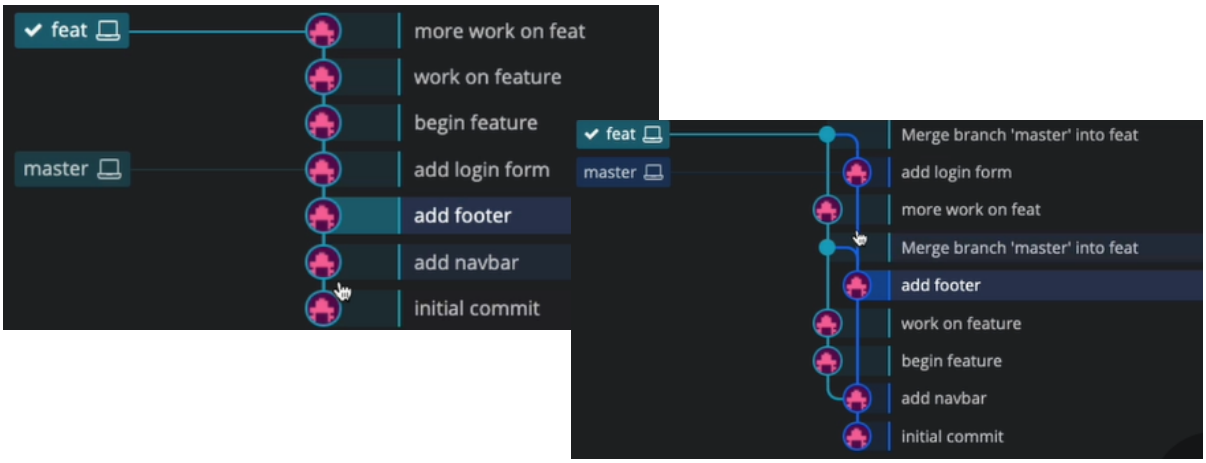
So this is not gonna screw up the master branch. Instead its gonna rewrite history by taking my feature branch commits and taking new ones that are based on the original and putting it at the tip of the master branch. Rebasing them at the tip of the master.



How many commits on my feature branch, the work is not gonna be destroyed. But the commits are going to re-created and they are gonna be add to the tip of the master branch.

We ended up with a linear structure.

```
RebaseDemo ❯ git rebase master                    feat
First, rewinding head to replay your work on top of
it...
Applying: begin feature
Applying: work on feature
Applying: more work on feat
RebaseDemo ❯                                      feat
```



bottom 4 are master and top 3 are from feat

After rebase

```
6320aa5 (HEAD -> feat) more work on feat
0e70bf5 work on feature
9071197 begin feature
c3ec574 (master) add login form
e19d409 add footer
34f6bd0 add navbar
2be74d9 initial commit
(END)
```

We see the commit hash is changed, but the commit message are still the same and the metadata.

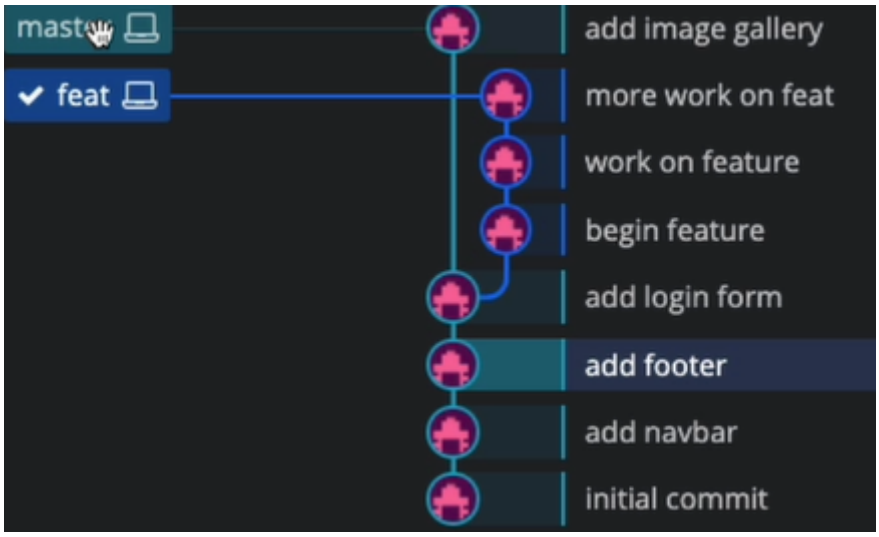
```
8b93760 (HEAD -> feat) Merge branch 'master' into fe
at
c3ec574 (master) add login form
33d06bc more work on feat
d9c49b6 Merge branch 'master' into feat
e19d409 add footer
c2b44e6 work on feature
8894594 begin feature
34f6bd0 add navbar
2be74d9 initial commit
```

Before rebase

This tell us its not update to commit history. Its new history.

this one is mixed, gitkraken is much needed

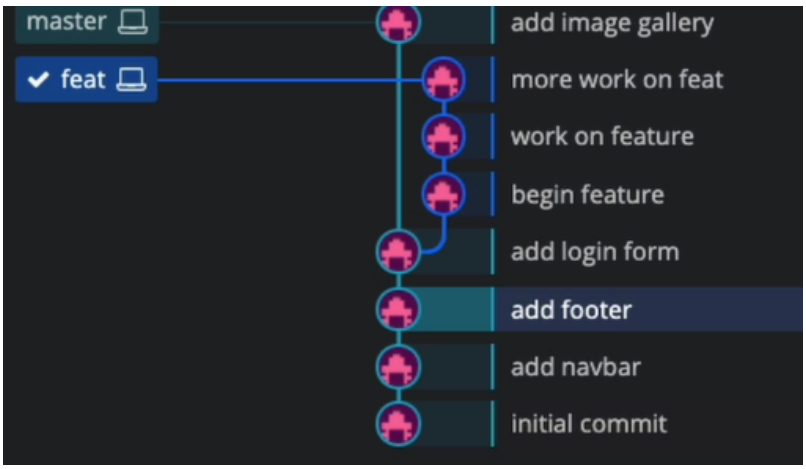
Note: colt shows us merging first and then rebasing.



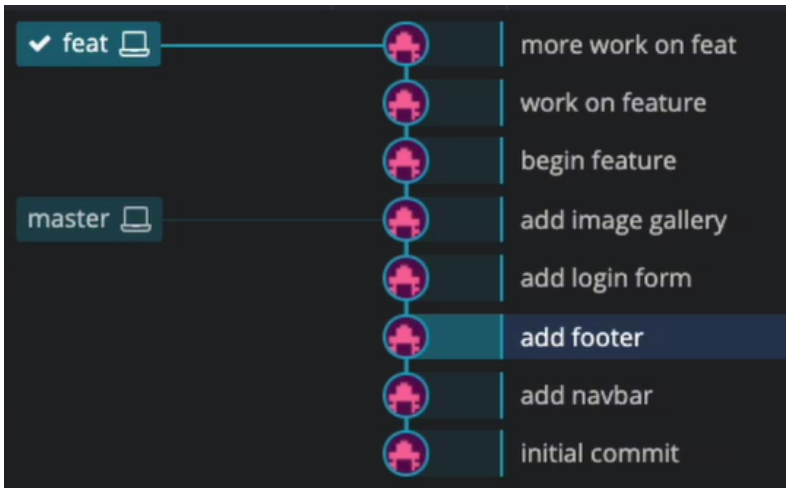

```
RebaseDemo ❯ git diff feat..master                    feat
```

```
index 981d74e..0000000
--- a/feature.txt
+++ /dev/null
@@ -1,2 +0,0 @@
-ONE
-TWO!
\ No newline at end of file
diff --git a/website.txt b/website.txt
index f8bd240..691df34 100644
--- a/website.txt
+++ b/website.txt
@@ -1,3 +1,4 @@
 NAVBAR ADDED!
 FOOTER ADDED!
-LOGIN FORM ADDED!
\ No newline at end of file
+LOGIN FORM ADDED!
+IMAGE GALLERY ADDED!
\ No newline at end of file
```

Here, we didn't use git merge, we used rebase.
Previosly we used to do:
git switch feat
git merge master
and if we wanna rebase.
git switch feat
git rebase master.




```
RebaseDemo ❯ git rebase master                    feat
First, rewinding head to replay your work on top of
it...
Applying: begin feature
Applying: work on feature
Applying: more work on feat
RebaseDemo ❯                                      feat
```



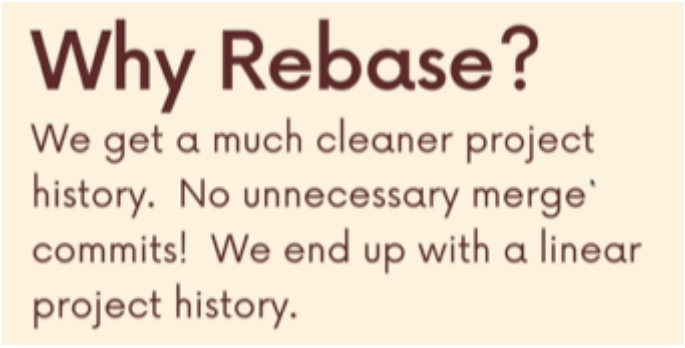
We not rebasing the master branch.

We are rebasing onto master branch.

So our feature branch commits are recreated, and all begins at the tip of the master and we end up with this nice linear structure.

when we rebase the branch, we are basing the branch on tip of another branch

The Golden Rule: When Not to Rebase



Why rebase instead of merge?

Its very clear what work a specific person did. So it doesn't matter bunch of 10 and 100 other commits on master branch, that they happen simultaneously over the course of development.

My feature branch commits are all located together at the tip. This makes it easier to for someone to review my commits and take a look at the code base and try to read whats going on.

In open source projects, with thousand of contributors and tons of work all the time. This makes it easier for some random person to go read the history, go try and understand what did this feature do.

Rebase makes total sense by kazim.

WARNING!

Never rebase commits that have been shared with others. If you have already pushed commits up to Github...DO NOT rebase them unless you are positive no one on the team is using those commits.

SERIOUSLY!

You do not want to rewrite any git history that other people already have. It's a pain to reconcile the alternate histories!

You only rebase commits that you only have in your machine and other people don't.

So your feature branches that you're working on, you don't want to rebase the master branch because people presumably have that master branch or the main branch they have the commits that you have.

So if you rewrite those commits and they're totally new, thats a problem when you try and push up. after u rebase, cuz ur collaborators have different history now

So you work on a feature branch, whatever you're doing in there, fixing a bug, adding a new feature, etc.

You can rebase your work from that branch onto the master branch or onto another branch. You not changing the master branch. You don't change the branch that you are rebasing onto.

We didn't change the master branch, we only change the feature branch. history

The branch that you are on when you call rebase is the one that is being rebased.

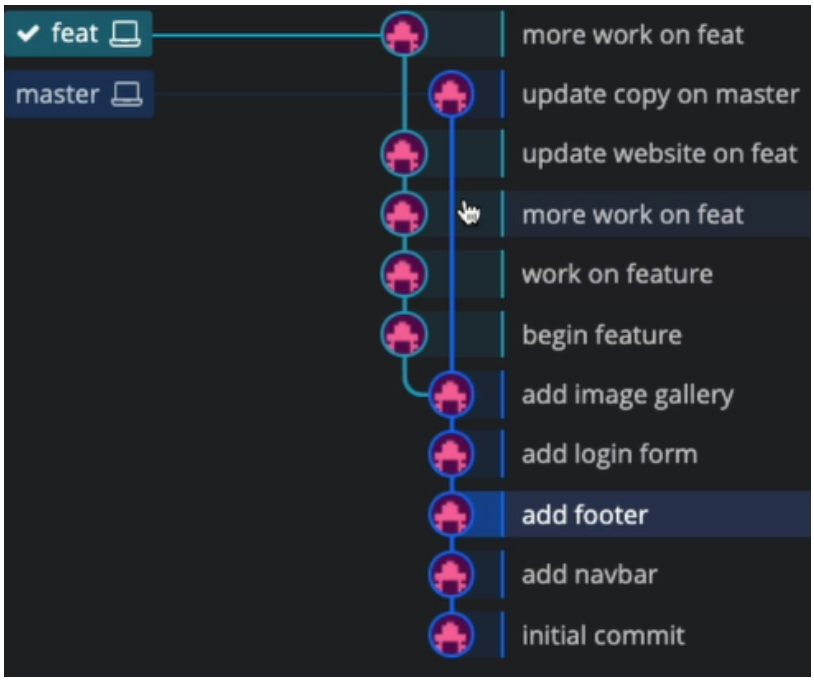
Goldren rule when not to:

So once you push the feature branch upm if somebody has pulled that down, if they have that history. If they working on it, then you don't wanna rebase.

Your company will tell you, if you are collaborating with people, you're rewriting history using rebase. If you don't know how to properly rebase that can be a big problem. If you are working at a company, they are going to make it very clear to you what the workflow is. And if they expect you to rebase everything or to never rebase and only or rebase in X situtions.

Dont rebase shared history and any commits ppl already have only rebase stuff that is yours and yours alone.

Handling Conflicts and Rebasing



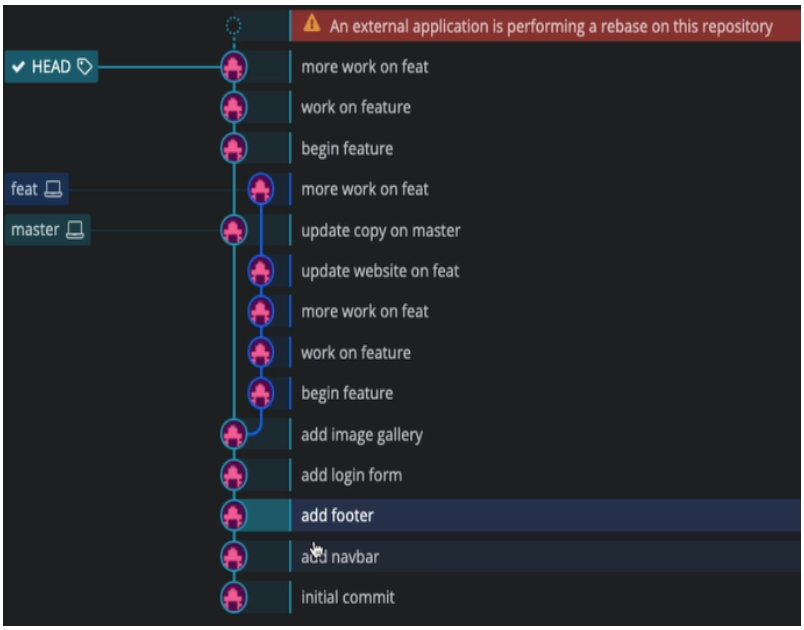
we want the feat branch on the tip of master branch.

```
RebaseDemo ❯ git rebase master                    feat
```

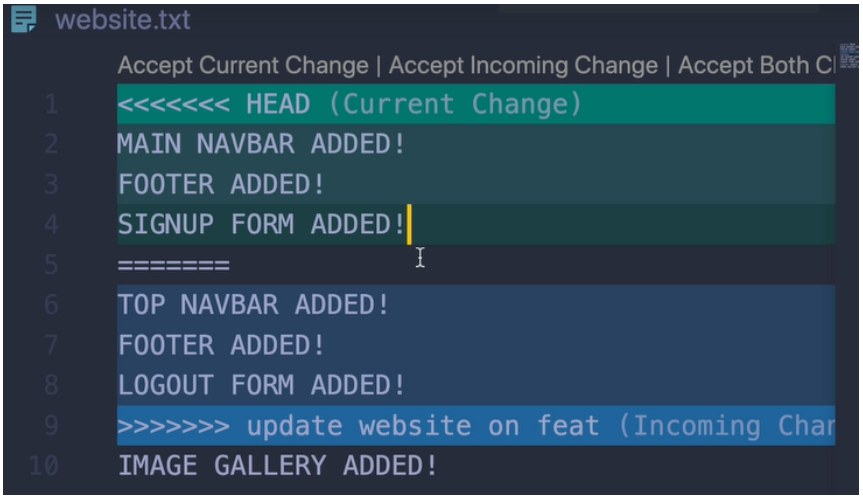
```
Applying: begin feature
Applying: work on feature
Applying: more work on feat
Applying: update website on feat
Using index info to reconstruct a base tree...
M       website.txt
Falling back to patching base and 3-way merge...
Auto-merging website.txt
CONFLICT (content): Merge conflict in website.txt
error: Failed to merge in the changes.
Patch failed at 0004 update website on feat
hint: Use 'git am --show-current-patch' to see the f
ailed patch
Resolve all conflicts manually, mark them as resolve
d with
"git add/rm <conflicted_files>", then run "git rebas
e --continue".
You can instead skip this commit: run "git rebase --
skip".
To abort and get back to the state before "git rebas
e", run "git rebase --abort".
```

Merge conflict

steps to follow




```
Using index info to reconstruct a base tree...
M       website.txt
Falling back to patching base and 3-way merge.
```




```
Resolve all conflicts manually, mark them as resolv
d with
"git add/rm <conflicted_files>", then run "git reba
e --continue".
You can instead skip this commit: run "git rebase -
skip".
To abort and get back to the state before "git reba
e", run "git rebase --abort".
```

```
RebaseDemo ❯ git status
rebase in progress; onto 4729c3c
You are currently rebasing branch 'feat' on '4729c3c'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
        both modified:   website.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

```
RebaseDemo ❯ git rebase --continue
Applying: update website on feat
Applying: more work on feat
RebaseDemo ❯ git status
On branch feat
nothing to commit, working tree clean
```

rebase is also an alternative to merge. so when u resolve conflicts. u make changes. so u need to add those conflicted files to ur staging area.
do u need to commit

Question on udemy by someone

Rebase resolving conflict: which commit got changed?

i think the top one. i haven't practice this out by myself. But yes the top one.

And ig we can view it as another commit. but it got amended with the recent commit on the feature branch that existed before we started rebasing.

--

we still have the date and time to know what order it goes.

we can use Gitkraken for that.

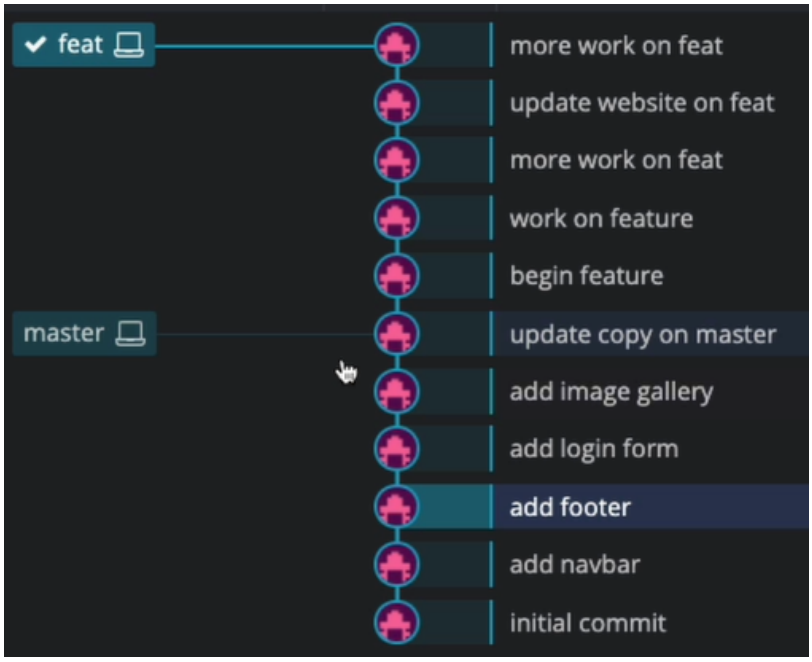
what can we do to avoid the conflict:

we could solve the conflict by merging and make an ammend. and then rebase would go without conflict.

basically all the commits from the feature branch then will sit on the main branch.

Considering other factors as well, what is the workflow for merging it to the main branch.

so the history rebase on top of the master branch.
rebase means we are changing the base.
so you have to be at somewhere before you change it to somewhere.



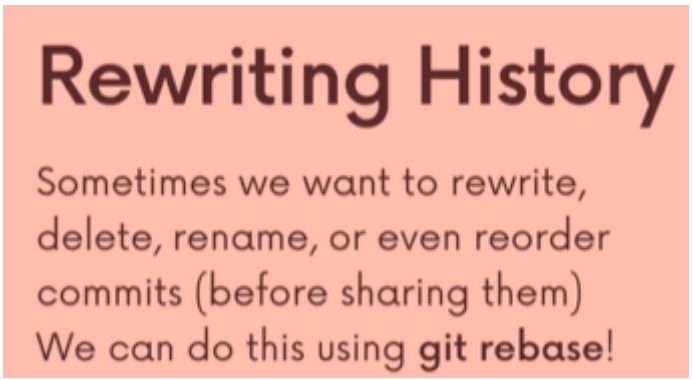
Cleaning up History with interactive rebase

We use it to clean up our git history. Basically to go back and rewrite commits.

Not just the previous commits but 10,50 far away.

We can rewrite commits, you can edit commits, we can do something called fixing up and squashing commits. We can even drop commits entirely.

Its fun to go back and edit history.



Rewrite history, edit commits, can change/edit commits n commit msgs.

We can change the contents of a commit. We can drop or delete an entire commit.

We can even re-order thing using git rebase.

work/commits that people already have.

We don't wanna rewrite history on ~~shared files~~. We don't wanna rewrite commits and make new commit hashes and screw everything up, if collaborators already have the work.

This is a common workflow, if you've been working on a branch. You have lots of commits on a feature you've been working on for a while. You kind have some half baked commits. Or some buggie thing or half complete commits.

If you want to go through and streamline that and combine commits down, rename them, that could be a nice thing to do b4 you share that code with anyone.

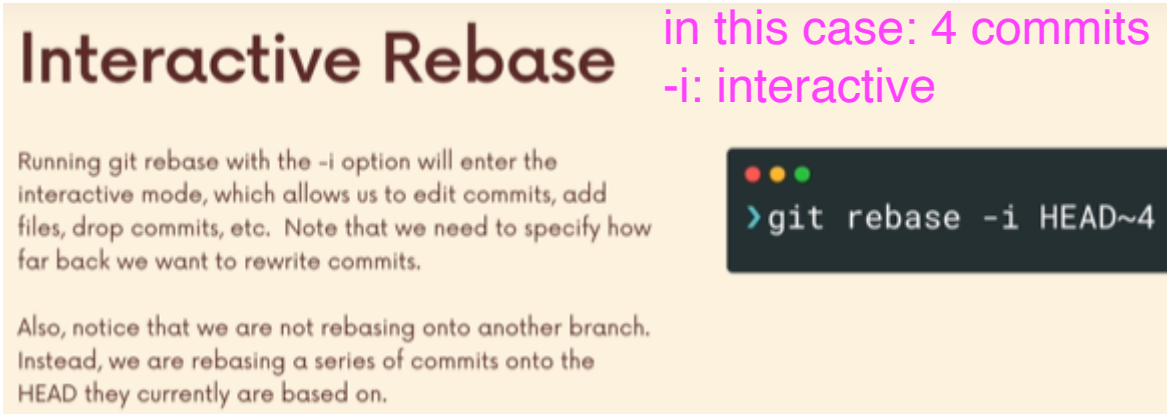
So its a step a lot of ppl take before, sort fo grand unveiling of their branch of their feature.

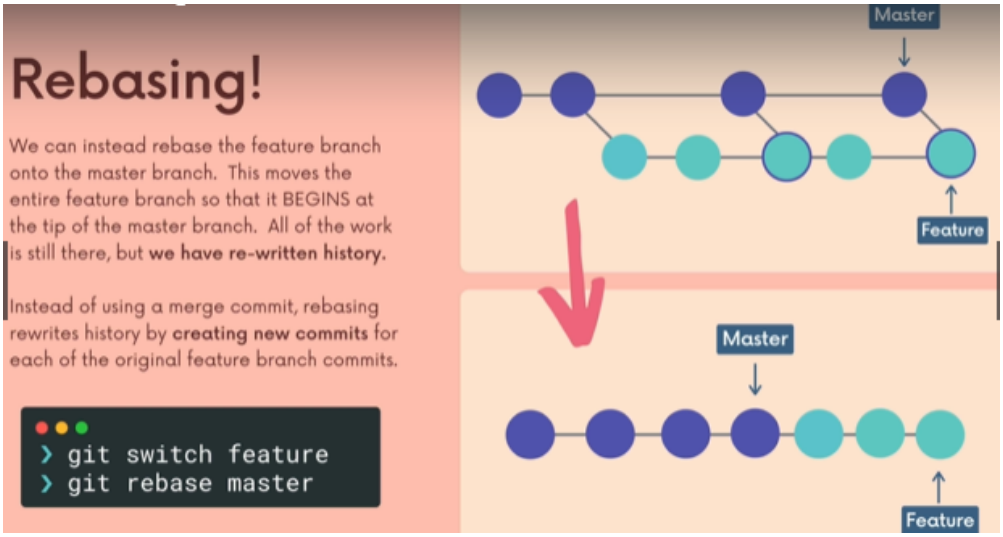
We use the git rebase cmd, typically we don't specify a branch to rebase onto.

Instead we just gonna rebase series of commits onto the head we currently based on. so, instead of rebasing onto the master branch, for e.g., or the main branch, we'll rebase on whatever branch we are on

So we gonna rebase myfeature where it currently is, I.e my feature.

Its just a way of recreating commits, every commit that we specify and we do have to provide a range. How far do we wanna go back? Recreate each one.





it went through all the commits on feature branch and it recreated it each one at the tip of the branch. Each new commits has a new hash, but it still have the same contents ,the same commit message . But they are distinct commits.

Using the interactive mode, is as you make a new commit, change the name or drop it.Dont make a new version of this one. it gives u a abiltiy to say

```
emo$ g
it log --oneline
3b23c17 (HEAD -> main, origin/main, origin/HEAD) Create README.md
2029e20 my cat made this commit
655204d fix another navbar typo
2a45e71 fix navbar typos
4ff2290 add top navbar
6e39a76 whoops forgot to add bootstrap js script
240827f add bootstrap
519aab6 add basic HTML boilerplate
cbee26b I added project files
```

This is a silly e.g, you're working on a project locally where you're working on your own and not paying attention to commit guidelines

You wanna rename the commit msg before you share that with somebody. Often when you doing some work locally on some project. You working on your own, you not paying attention to commit message guidelines.

```
-demo$ git log --oneline
3b23c17 (HEAD -> main, origin/main, origin/HEAD) Create README
2029e20 my cat made this commit
655204d fix another navbar typo
2a45e71 fix navbar typos
4ff2290 add top navbar
6e39a76 whoops forgot to add bootstrap js script
240827f add bootstrap
519aab6 add basic HTML boilerplate
cbee26b I added project files
0e19c7a initial commit
```

But then as you get ready to share this branch with people. and somebody who maintains a big open source project or ur boss, you might want to go through and rename commits to conform to some pattern that the project is asking you to do.

I wanna combine that into one commit. cuz i'm doing the same thing

```
-demo$ git log --oneline
3b23c17 (HEAD -> main, origin/main, origin/HEAD) Create README.md
2029e20 my cat made this commit
655204d fix another navbar typo
2a45e71 fix navbar typos
4ff2290 add top navbar
6e39a76 whoops forgot to add bootstrap js script
240827f add bootstrap
519aab6 add basic HTML boilerplate
cbee26b I added project files
0e19c7a initial commit
```

i wanna make this into one, just a complete navbar

Also I wanna drop the comit "my cat made this commit"

Interactive Rebase

Running git rebase with the -i option will enter the interactive mode, which allows us to edit commits, add files, drop commits, etc. Note that we need to specify how far back we want to rewrite commits.

Also, notice that we are not rebasing onto another branch. Instead, we are rebasing a series of commits onto the HEAD they currently are based on.

```
>git rebase -i HEAD~4
```

i want to go through and clean some of these commits, condense them, maybe drop them , rename them, before anyone sees them


```
...l/Desktop/Learning Git/interactive-rebase-demo
pick cbee26b I added project files
pick 519aab6 add basic HTML boilerplate
pick 240827f add bootstrap
pick 6e39a76 whoops forgot to add bootstrap js sc
pick 4ff2290 add top navbar
pick 2a45e71 fix navbar typos
pick 655204d fix another navbar typo
pick 2029e20 my cat made this commit
pick 3b23c17 Create README.md
```

we can use git rebase -i Head~4
and specify how far back we want to go
and again, i'm not specifying a branch to rebase onto. IOW, redo some number of commits on this branch

```
# Rebase 0e19c7a..3b23c17 onto 2029e20 (9 commands)
#
# Commands: git rebase -i Head~9. generally we wanna minimize the amount of rewriting we do.
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

It shows me all the commits but not every, only the one i selected.

```
pick cbee26b I added project files
pick 519aab6 add basic HTML boilerplate
pick 240827f add bootstrap
pick 6e39a76 whoops forgot to add bootstrap js script
pick 4ff2290 add top navbar
pick 2a45e71 fix navbar typos
pick 655204d fix another navbar typo
pick 2029e20 my cat made this commit
pick 3b23c17 Create README.md
```

there are 10 commits in total, we see only 9 and the first commit's message was "initial commit"

Notice that the order is reversed and we don't see the "initial commit"

Basically we running pick command on all the commits.

Order

```
reword cbee26b I added project files
pick 519aab6 add basic HTML boilerplate
pick 240827f add bootstrap
pick 6e39a76 whoops forgot to add bootstrap js script
pick 4ff2290 add top navbar
pick 2a45e71 fix navbar typos
pick 655204d fix another navbar typo
pick 2029e20 my cat made this commit
pick 3b23c17 Create README.md
```

Then save.

But if i change what these are, if i drop something, rename, edit, it's gonna go through and order and thake this commit and follow this instructions.

so basically u only tell the rebase command(pick,reword,edit,fixup,drop). and save

```
reword cbee26b I added project files
pick 519aab6 add basic HTML boilerplate
pick 240827f add bootstrap
pick 6e39a76 whoops forgot to add bootstrap js scrip
pick 4ff2290 add top navbar
pick 2a45e71 fix navbar typos
pick 655204d fix another navbar typo
pick 2029e20 my cat made this commit
pick 3b23c17 Create README.md
```

Save and close the editor.

by itself

Then, editor will open to write new commit message.

```
I added project files

# Please enter the commit message for your changes. Lines s
# with '#' will be ignored, and an empty message aborts the
#
# Author:    Colt Steele <5498438+Colt@users.noreply.github
# Date:      Tue Feb 9 13:58:21 2021 -0800
#
# interactive rebase in progress; onto 0e19c7a
# Last command done (1 command done):
#    reword cbee26b I added project files
# Next commands to do (8 remaining commands):
#    pick 519aab6 add basic HTML boilerplate
#    pick 240827f add bootstrap
# You are currently editing a commit while rebasing branch
#
# Changes to be committed:
#       new file:   app.css
```

```
interactive-rebase-demo ❯ git rebase -i HEAD~9                    my-feat
hint: Waiting for your editor to clhint: Waiting for your editor to cl[d
etached HEAD 83cb8ad] add project files
 Date: Tue Feb 9 13:58:21 2021 -0800
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 app.css
 create mode 100644 index.html
Successfully rebased and updated refs/heads/my-feat.
```

```
a3037e0 (HEAD -> my-feat) Create README.md
daacc7e my cat made this commit
35a6c49 fix another navbar typo
4f6d68d fix navbar typos
71b0efb add top navbar
4c6cb28 whoops forgot to add bootstrap js script
21acd17 add bootstrap
b35e3a4 add basic HTML boilerplate
83cb8ad add project files
0e19c7a initial commit
```

By rewording, all commit is changed starting from the commit which is changed.

```
2029e20 a3037e0 (HEAD -> my-feat) Create README.md
655204d daacc7e my cat made this commit
2a45e71 35a6c49 fix another navbar typo
4ff2290 4f6d68d fix navbar typos
6e39a76 71b0efb add top navbar
240827f 4c6cb28 whoops forgot to add bootstrap js script
519aab6 21acd17 add bootstrap
cbee26b b35e3a4 add basic HTML boilerplate
0e19c7a 83cb8ad add project files
        0e19c7a initial commit
```

I added project files

we see all the 9 commit hash are different now, except the initial commit.
starting from the commit that was rewritten. since “add project files” commit changed cuz of reword.
so every subsequent commit hash change, cuz every commits takes into account the parent commit.

Fixing and Squashing Commits with interactive Rebase

```
-demo$ git log --oneline
3b23c17 (HEAD -> main, origin/main, origin/HEAD) Create README
2029e20 my cat made this commit
655204d fix another navbar typo
2a45e71 fix navbar typos
4ff2290 add top navbar
6e39a76 whoops forgot to add bootstrap js script
240827f add bootstrap
519aab6 add basic HTML boilerplate
cbee26b I added project files
0e19c7a initial commit
```

But the msg are the same, the commit are the same,
we didn’t rebase onto another branch.

```
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log
message
```

colt prefer fixup, but i think squash is also good or i say better.

since the history is upside down. and now we gonna fixup which
will take the prev commit and mend it into one.
with the prev commit msg be the one which survives

```
pick 83cb8ad add project files
pick b35e3a4 add basic HTML boilerplate
pick 21acd17 add bootstrap
fixup 4c6cb28 whoops forgot to add bootstrap js script
pick 71b0efb add top navbar
pick 4f6d68d fix navbar typos
pick 35a6c49 fix another navbar typo
pick daacc7e my cat made this commit
pick a3037e0 Create README.md
```

```
63ff2f1 (HEAD -> my-feat) Create README.md
592b3b3 my cat made this commit
e4f2eac fix another navbar typo
507cf59 fix navbar typos
de10e00 add top navbar
3b0d8a6 add bootstrap
b35e3a4 add basic HTML boilerplate
83cb8ad add project files
0e19c7a initial commit
```

all the commits hash after the add bootstrap will be new

Everything from where the changes made to the freshest change all hashes are change.

```
interactive-rebase-demo ❯ git rebase -i HEAD~9                    my-feat
fatal: invalid upstream 'HEAD~9'
interactive-rebase-demo ❯ git rebase -i HEAD~8                    my-feat
hint: Waiting for your editor to close the file...
```

```
pick 83cb8ad add project files
pick b35e3a4 add basic HTML boilerplate
pick 3b0d8a6 add bootstrap
pick de10e00 add top navbar
fixup 507cf59 fix navbar typos
fixup e4f2eac fix another navbar typo
pick 592b3b3 my cat made this commit
pick 63ff2f1 Create README.md
```

```
interactive-rebase-demo ❯ git rebase -i HEAD~8                    my-feat
Successfully rebased and updated refs/heads/my-feat.
interactive-rebase-demo ❯ g                                       my-feat
```

Github shows what actually changed between subsequent commits.

```
a354764 (HEAD -> my-feat) Create README.md
923b820 my cat made this commit
45493e7 add top navbar
3b0d8a6 add bootstrap
b35e3a4 add basic HTML boilerplate
83cb8ad add project files
0e19c7a initial commit
```

do read the prev page commits to understand this one.
Basically the older win eats the newone.

Dropping Commits with Interactive Rebase

```
a354764 (HEAD -> my-feat) Create README.md
923b820 my cat made this commit
45493e7 add top navbar
3b0d8a6 add bootstrap
b35e3a4 add basic HTML boilerplate
83cb8ad add project files
0e19c7a initial commit
(END)
```

```
interactive-rebase-demo ❯ git rebase -i HEAD~2                    my-feat
```

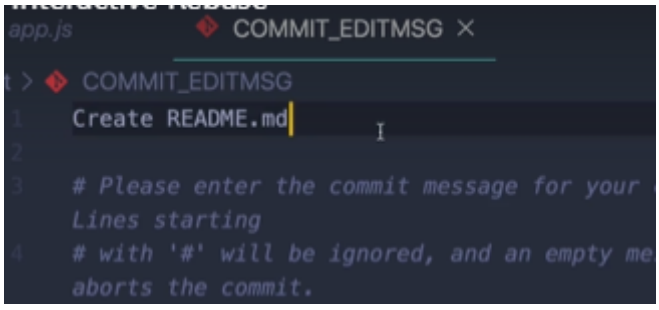
```
drop 923b820 my cat made this commit
pick a354764 Create README.md
```

```
interactive-rebase-demo ❯ git rebase -i HEAD~2                    my-feat
hint: Waiting for your editor to clSuccessfully rebased and updated ref
/heads/my-feat.
```

```
491396e (HEAD -> my-feat) Create README.md
45493e7 add top navbar
3b0d8a6 add bootstrap
b35e3a4 add basic HTML boilerplate
83cb8ad add project files
0e19c7a initial commit
(END)
```

The commit disappeared with its contents.

```
interactive-rebase-demo ❯ git commit --amend                      my-feat
```



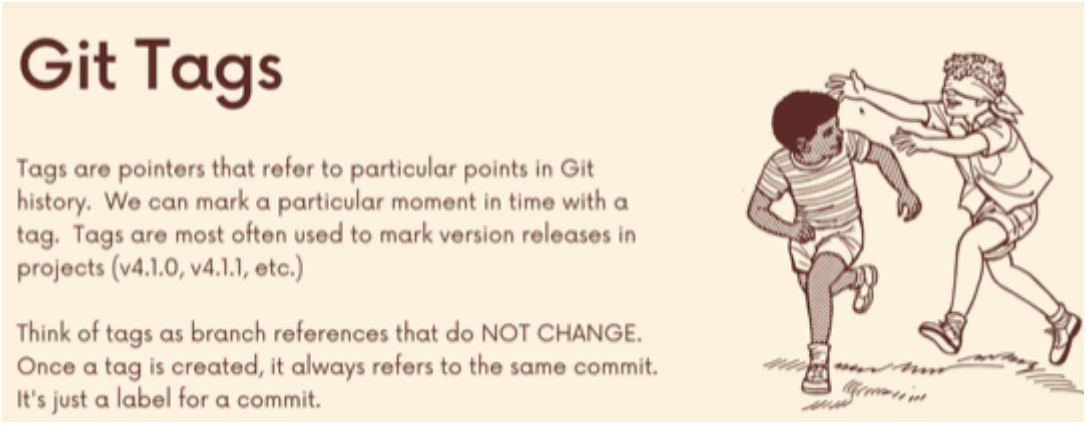
here colt is talking about rewording. but if its the latest commit, we can instead amend which will
let us add new files or changes and even modify the commit msg.

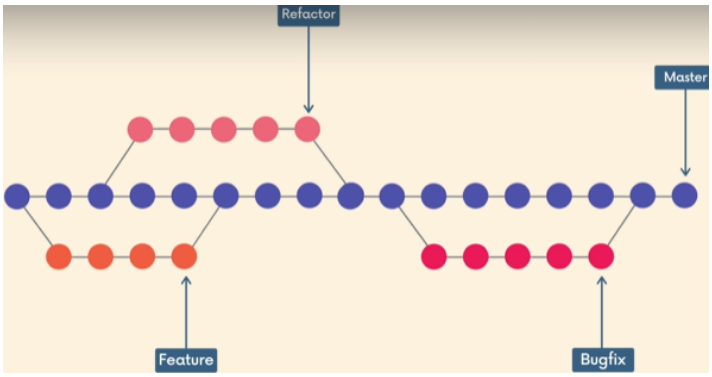
```
393aee2 (HEAD -> my-feat) create README.md
45493e7 add top navbar
3b0d8a6 add bootstrap
b35e3a4 add basic HTML boilerplate
83cb8ad add project files
0e19c7a initial commit
```

Git Tags: Marking important moments in histories

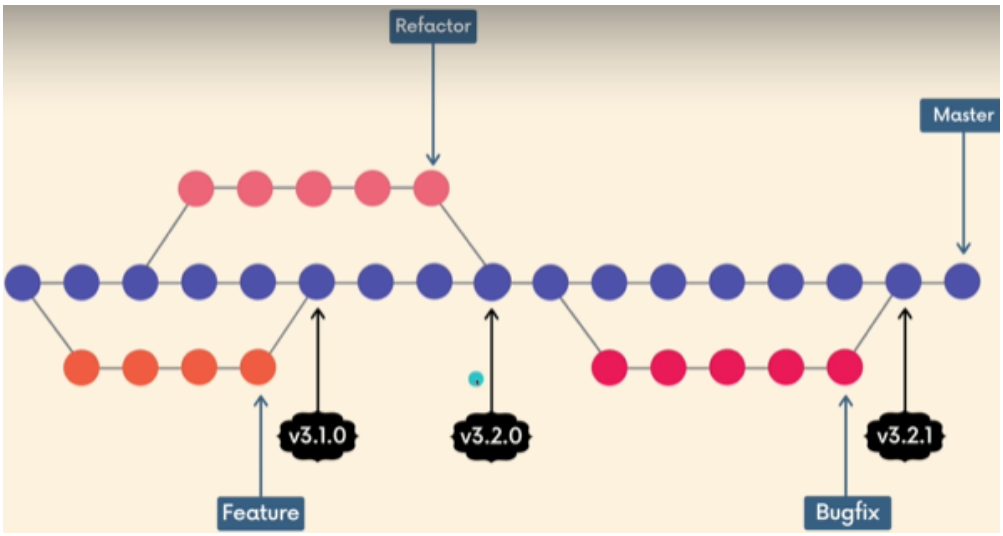
Most often tags are used to mark different releases, different versions or projects.

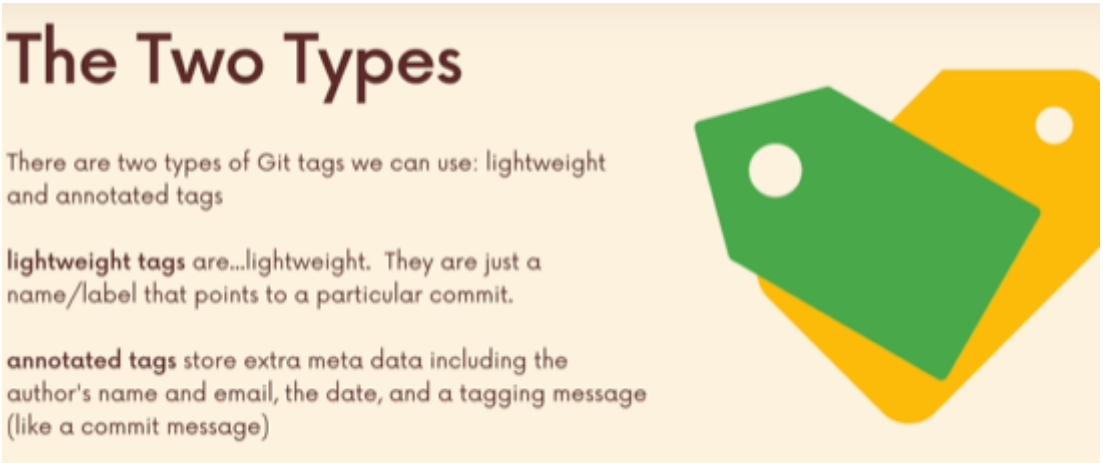
The Idea behind Git Tag





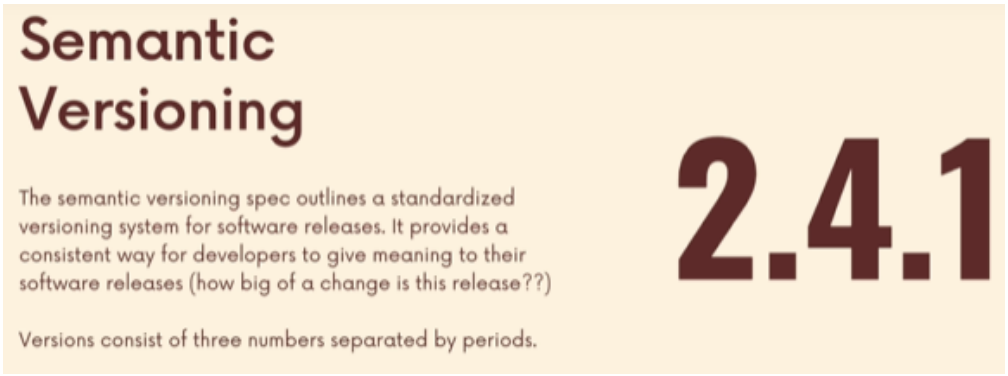
Think of as a sticky note on commits.





Annotated Tags are generally preferred in a lot of big projects. They will ask you to use annotated Tags and not lightweighted Tags.

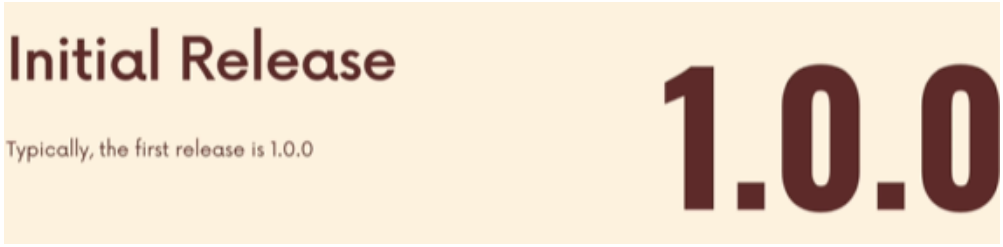
Semantic Versioning

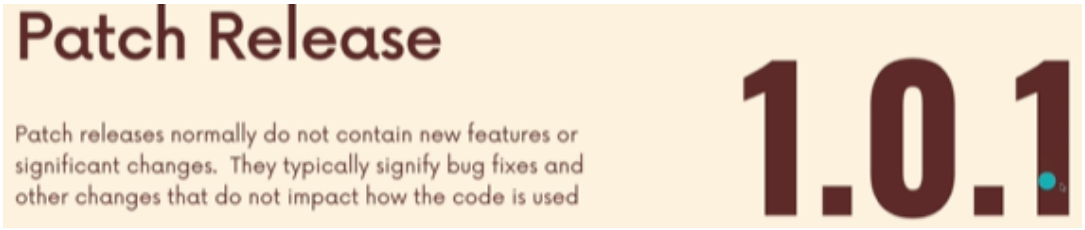


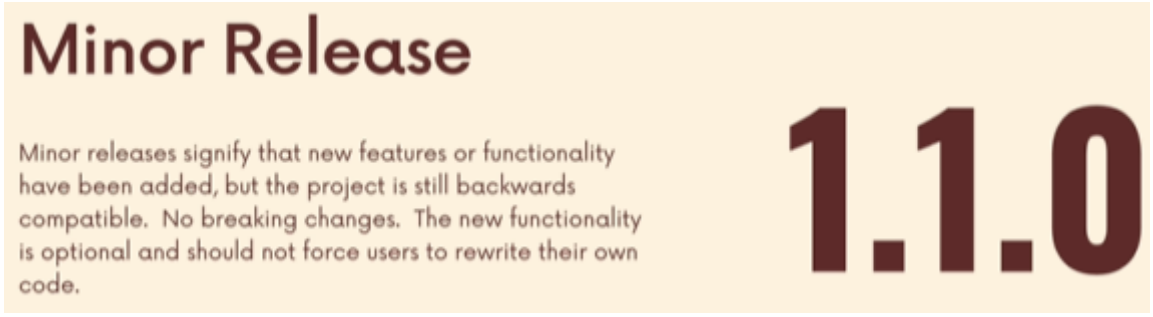
The most of common use case of tags is versioning and marking release point for for some piece of software.

Semantic Versioning is a protocol,set of rules,requirements that dictate how version number is assigned and incremented.

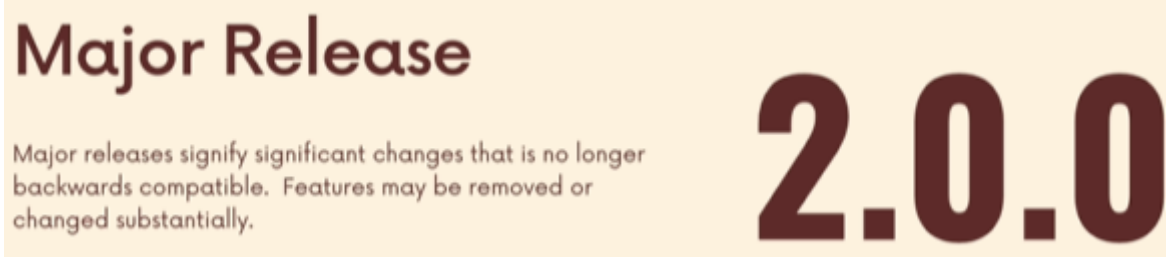
Semantic Versioning:
The idea is just to layout set of rules for making versioning transparent and consistent meaningful. So that these release number as meaning encoded in those three digits.







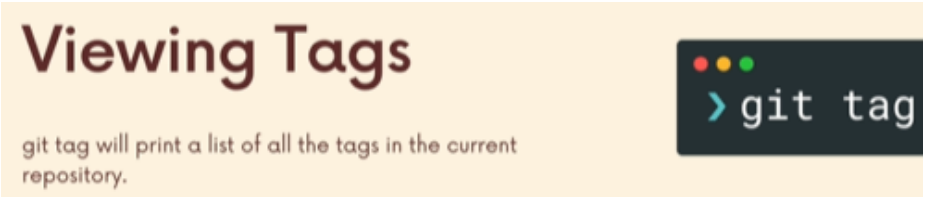
Whenever there is a minor release you need to reset the patch number back to zero.

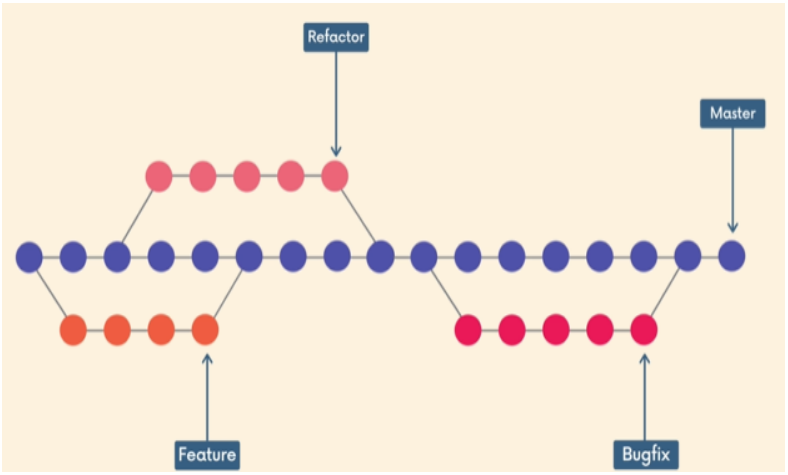


It makes it clear to any users or consumers of some piece of software, how significant a change is, in a new version.

Which is very important when you relying on some piece of software or library as dependencies for your own software you building.

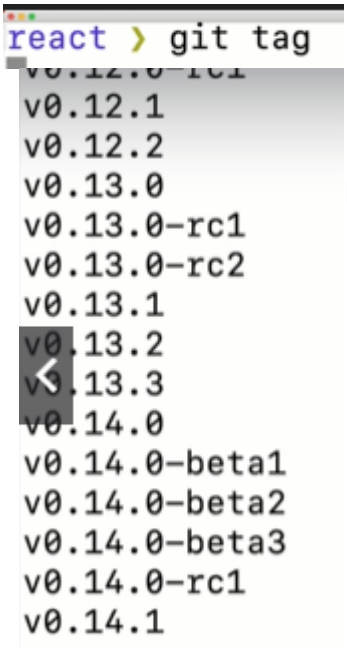
Viewing & Searching Tags



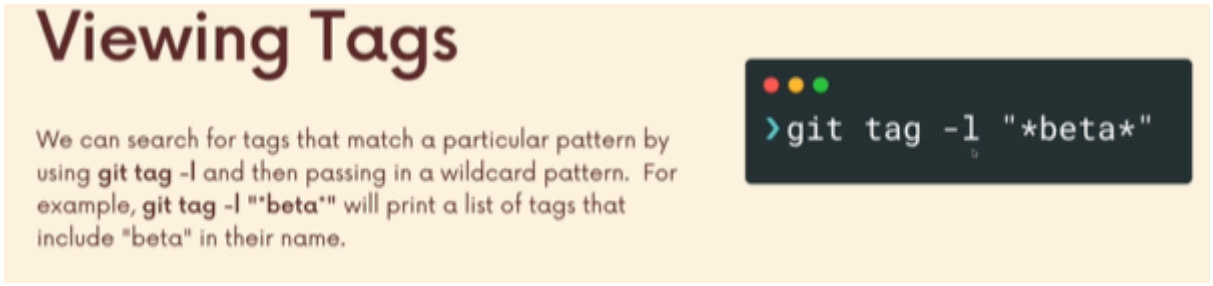


Static reference.

Concept of tag is to mark or highlight some point in history, have that as a static reference. Not move it until we explicitly want it to move it.



Each refer to a particular point in time, commit.

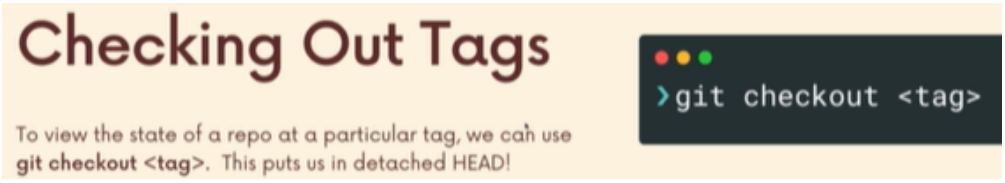


-l is implicit

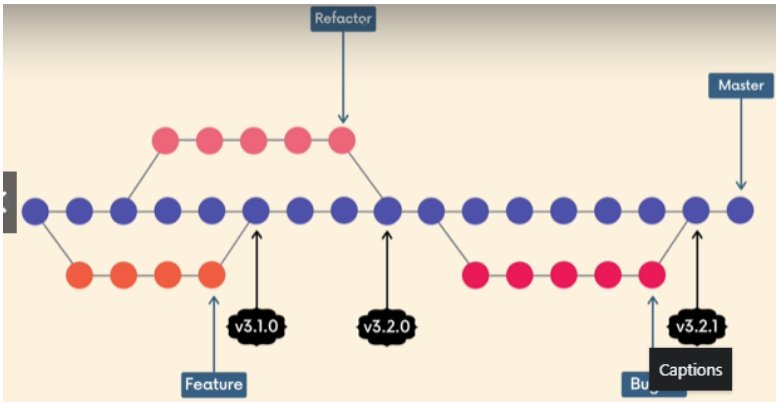
```
react ❯ git tag -l "v17*"
v17.0.0
v17.0.1
(END)
```

```
react ❯ git tag  "*beta*"                                    master
fatal: '*beta*' is not a valid tag name.
```

Comparing Tags With Git Diff



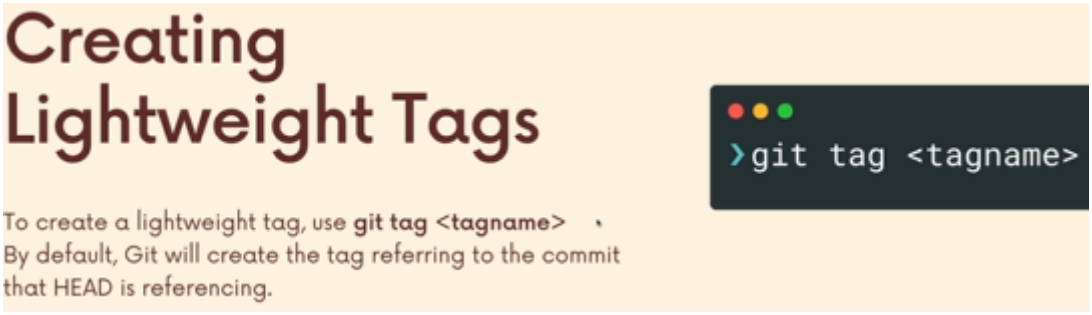
Tag is referring to a particular commit not a branch.



When we gonna checkout a tag, we will be at detached head.

```
react ❯ git tag                                                    mast
react ❯ git checkout 15.3.1                                        mast

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

If you want to create a new branch to retain commits you create,
you may
do so (now or later) by using -c with the switch command. Example
:

  git switch -c <new-branch-name>

react ❯ git switch -c BRANCH_FROM_TAG                              HEAD
Switched to a new branch 'BRANCH_FROM_TAG'

react ❯ git tag                                            BRANCH_FROM_TAG
react ❯ git diff v17.0.0 v17.0.1                           BRANCH_FROM_TAG
```

Creating Lightweight Tags



Creating Lightweight Tags

To create a lightweight tag, use **git tag <tagname>**
By default, Git will create the tag referring to the commit that HEAD is referencing.

```
>git tag <tagname>
```

```
react ❯ git tag v17.0.2                                    master
```

```
07027f93e (HEAD -> master) update README
07934efc0 (tag: v17.0.2) fix typo in README
1a7472624 (origin/master, origin/HEAD) Add
o the host config (#20809)
696e736be Warn if static flag is accidental
```

```
react ❯ git tag v17.0.3
```

```
v16.9.0
v16.9.0-alpha.0
v16.9.0-rc.0
v17.0.0
v17.0.1
v17.0.2
v17.0.3
(END)
```

Annotated Tags

```
react ❯ git tag -a v17.1.0
```

README.md | TAG_EDITMSG ●

TAG_EDITMSG

```
#
# Write a message for tag:
#   v17.1.0
# Lines starting with '#' will be ignored.
this is more stuff about my tag!! it's an annotated tag :)
```

```
v16.9.0
v16.9.0-alpha.0
v16.9.0-rc.0
v17.0.0
v17.0.1
v17.0.2
v17.0.3
v17.1.0
```

Annotated Tags

Use **git tag -a** to create a new annotated tag. Git will then open your default text editor and prompt you for additional information.

Similar to git commit, we can also use the **-m** option to pass a message directly and forgo the opening of the text editor

```
>git tag -a <tagname>
```

```
react ❯ git show v17.1.0
```

To view the metadata.

```
tag v17.1.0
Tagger: Colt Steele <5498438+Colt@users.noreply.github.com>
Date:   Fri Feb 12 14:30:20 2021 -0800

this is more stuff about my tag!! it's an annotated tag :)

commit 796f12225a80ef10f46a4ebcd51a6ba3554152e5 (HEAD -> master
tag: v17.1.0)
Author: Colt Steele <5498438+Colt@users.noreply.github.com>
Date:   Fri Feb 12 14:29:55 2021 -0800

    add new feature

diff --git a/NEW_FEATURE.js b/NEW_FEATURE.js
new file mode 100644
index 000000000..31c9f9045
--- /dev/null
.
```

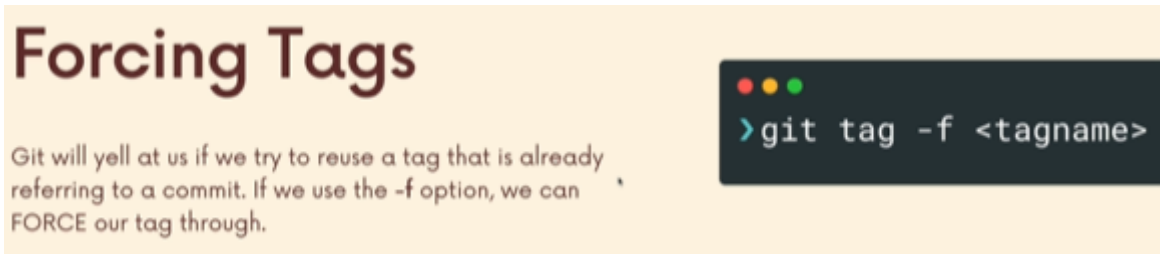
```
commit 89b610969d70d788f8c9769e3fa5b0044f5737ab (tag: v17.0.0, or
igin/17.0.0-dev)
Author: Dan Abramov <dan.abramov@me.com>
Date:   Tue Oct 20 21:29:42 2020 +0100

    Bump versions for 17

diff --git a/packages/create-subscription/package.json b/packages
/create-subscription/package.json
index 4677c731b..4342325d5 100644
--- a/packages/create-subscription/package.json
+++ b/p
@@ -1,
 {
   "nam
   "des                                                    urc
es inside React components",
```

As I'm editing this...I'm realizing this is a lightweight tag.
Notice the lack of "tagger", the tag date & message
All this shows is the commit info, no tag metadata

Tagging Previous Commits

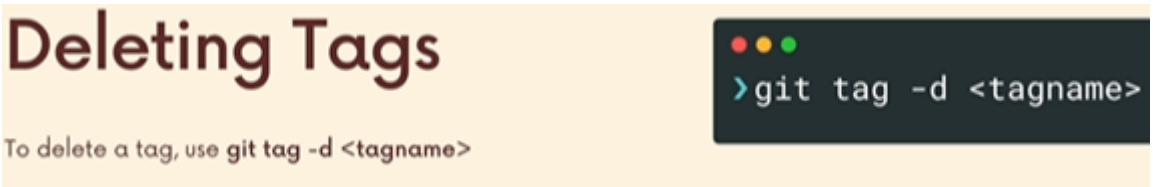

```
react › git log --oneline
react › git tag mytag 67e841982
```

Replacing Tags With Force




```
react › git log --oneline
react › git tag v17.0.3 696e736be
fatal: tag 'v17.0.3' already exists
react › git log --oneline
react › git tag v17.0.3 696e736be -f
Updated tag 'v17.0.3' (was 07027f93e)
```

Deleting Tags




```
react › git tag deleteme
react › git log --oneline
react › git tag -d deleteme
Deleted tag 'deleteme' (was 796f12225)
react › git log --oneline
```

IMPORTANT: Pushing Tags

Pushing Tags

By default, the **git push** command doesn't transfer tags to remote servers. If you have a lot of tags that you want to push up at once, you can use the --tags option to the **git push** command. This will transfer all of your tags to the remote server that are not already there.

```
›git push --tags
```

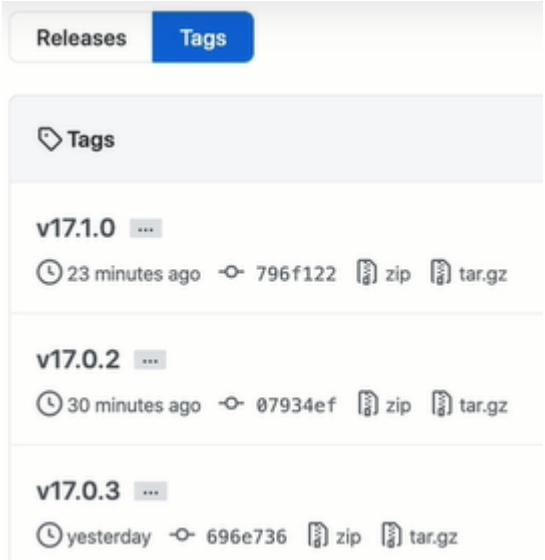
Sharing Tags

By default, the `git push` command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run `git push origin <tagname>`.

```
$ git push origin v1.5
```

```
react › git push origin mytag                                    master
remote: Permission to facebook/react.git denied to Colt.
fatal: unable to access 'https://github.com/facebook/react.git/':
 The requested URL returned error: 403
```

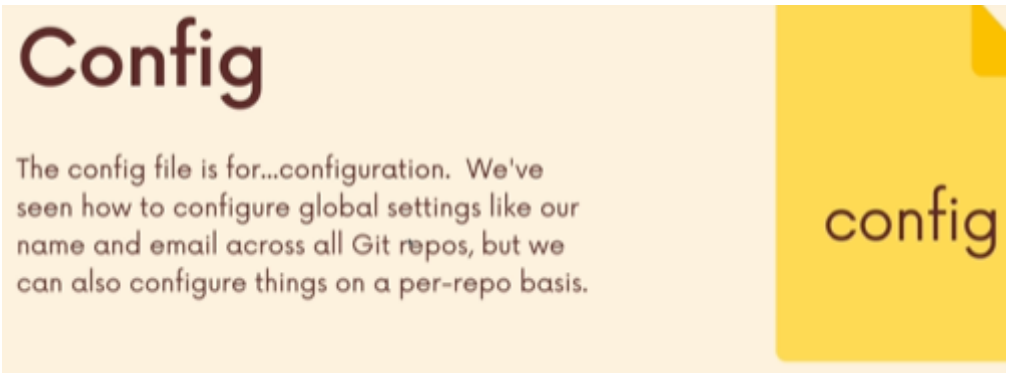
```
react › git push colt  mytag
Total 0 (delta 0), reused 0 (delta 0)
To github.com:Colt/tags-demo.git
 * [new tag]             mytag -> mytag
react › git push colt --tags
```



Git Behind the Scenes : Hashing and Objects

Working with the local file


```
.git ❯ ls
COMMIT_EDITMSG  description    info          packed-refs
HEAD            hooks          logs          refs
config _        index          objects
```



With git, there are multiple locations, from where we can configure settings. We can do it globally.

We can do system wide , globally.

Or local per repository basis.

So in every repositroy > .git > .config.

Anything u do in this folder will be local to this repository.

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = https://github.com/facebook/react.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[remote "colt"]
    url = git@github.com:Colt/tags-demo.git
    fetch = +refs/heads/*:refs/remotes/colt/*
```

In this folder you add git aliases.

```
.git ❯ git config user.name
Colt Steele
.git ❯ git config user.name "askdjhasd"
```

When I do the above, I do it globally.

```
.git ❯ git config --local user.name "chicken little"
.git ❯
```

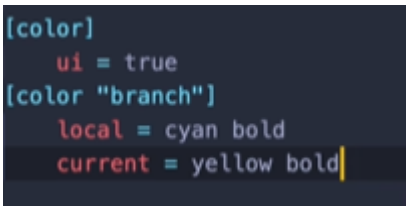
```
sahilsyed@DESKTOP-7VRCIK9:/mnt/c/Users/Dell/Desktop/Learning Git/interac
-demo$ git config -l
user.name=kazim
user.email=kazimsyed9911@gmail.com
email.name=kazimsyed9911@gmail.com
credential.helper=store
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
remote.origin.url=https://github.com/Colt/interactive-rebase-demo.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.main.remote=origin
```

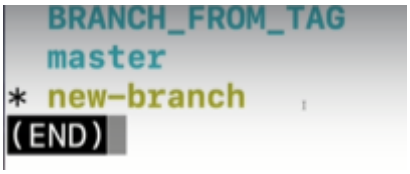
Check out the git-scm.com documentation.

```
  BRANCH_FROM_TAG
* master
(END)
```

```
[color]
    ui = true
[color "branch"]
    local = cyan bold
```

```
  BRANCH_FROM_TAG
  master
* new-branch
(END)
```





color

The value for a variable that takes a color is a list of colors (at most two, one for foreground and one for background) and attributes (as many as you want), separated by spaces.

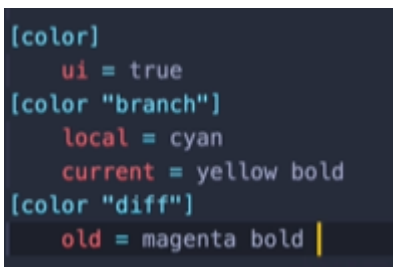
The basic colors accepted are `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` and `white`. The first color given is the foreground; the second is the background. All the basic colors except `normal` have a bright variant that can be specified by prefixing the color with `bright`, like `brightred`.

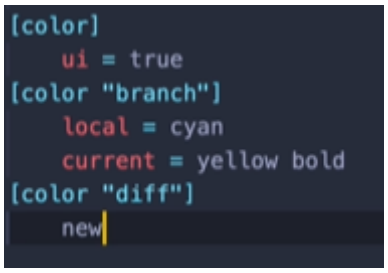
Colors may also be given as numbers between 0 and 255; these use ANSI 256-color mode (but note that not all terminals may support this). If your terminal supports it, you may also specify 24-bit RGB values as hex, like `#ff0ab3`.

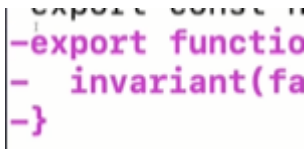
The accepted attributes are `bold`, `dim`, `ul`, `blink`, `reverse`, `italic`, and `strike` (for crossed-out or "strikethrough" letters). The position of any attributes with respect to the colors (before, after, or in between), doesn't matter. Specific attributes may be turned off by prefixing them with `no` or `no-` (e.g., `noreverse`, `no-ul`, etc).

An empty color string produces no color effect at all. This can be used to avoid coloring specific elements without disabling color entirely.

For git's pre-defined color slots, the attributes are meant to be reset at the beginning of each item in the colored output. So setting `color.decorate.branch` to `black` will paint that branch name in a plain `black`, even if the previous thing on the same output line (e.g. opening parenthesis before the list of branch names in `log --decorate` output) is set to be painted with `bold` or some other attribute. However, custom log formats may do more complicated and layered coloring, and the negated forms may be useful there.







You can change text editor locally for this repo.

Inside Git: The Refs Directory

refs for references.

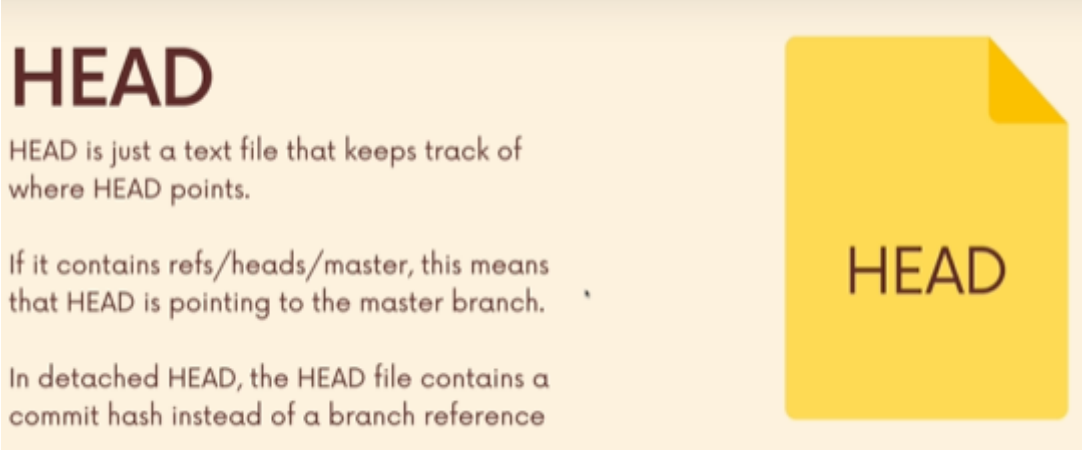
```
├── COMMIT_EDITMSG
├── HEAD
├── ORIG_HEAD
├── branches
├── config
├── description
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── fsmonitor-watchman.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── pre-merge-commit.sample
│   ├── pre-push.sample
│   ├── pre-rebase.sample
│   ├── pre-receive.sample
│   ├── prepare-commit-msg.sample
│   └── update.sample
├── index
├── info
│   └── exclude
├── logs
│   ├── HEAD
│   └── refs
│       ├── heads
│       │   ├── Sahil
│       │   ├── Sania
│       │   ├── kazim
│       │   └── master
│       └── remotes
│           └── origin
│               ├── master
│               └── sania
```

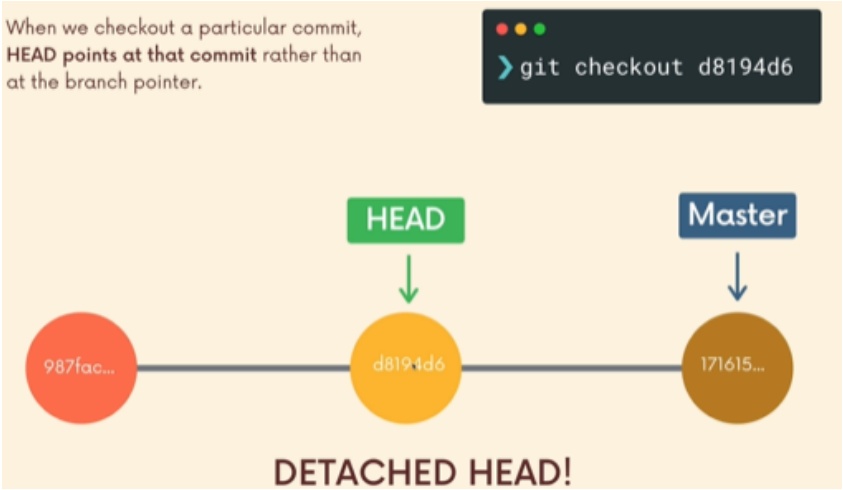
```
│   ├── info
│   └── pack
└── refs
    ├── heads
    │   ├── Sahil
    │   ├── kazim
    │   ├── master
    │   └── sania
    ├── remotes
    │   └── origin
    │       ├── master
    │       └── sania
    └── tags

78 directories, 104 files
```

```
├── objects
│   ├── 03
│   │   └── 5dc548450f2a2b6a0f0eaf7bce9e13
```

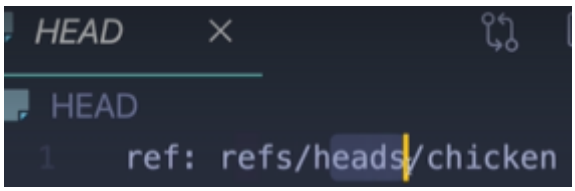
```
react ❯ git fetch                                                    chick
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Total 24 (delta 19), reused 19 (delta 19), pack-reused 5
Unpacking objects: 100% (24/24), 4.81 KiB | 234.00 KiB/s, done.
From https://github.com/facebook/react
   1a7472624..e2fd460cc  master     -> origin/master
```


Inside Git: The Head file

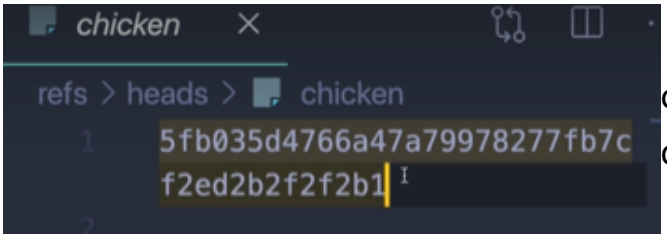




There is a difference between Head and Heads

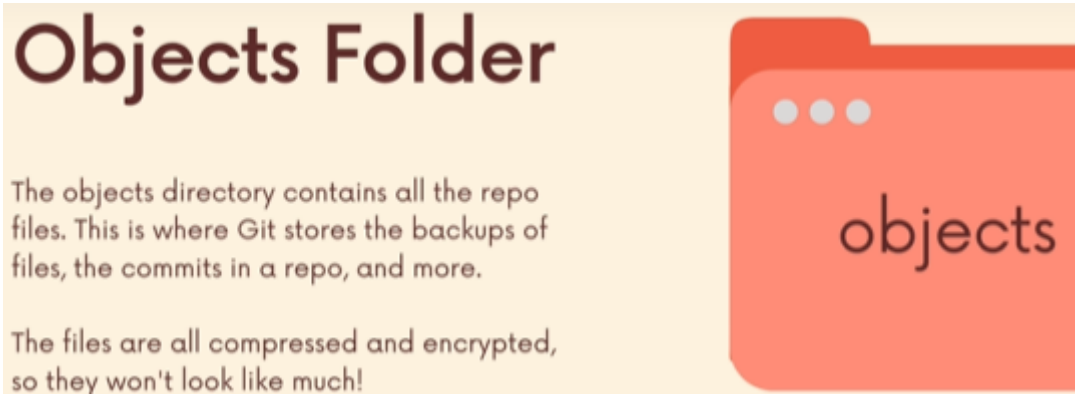


This is, where we at.



Head always point to the newest commit on a branch. Else we at detached head.

Inside Git: The Objects Folder

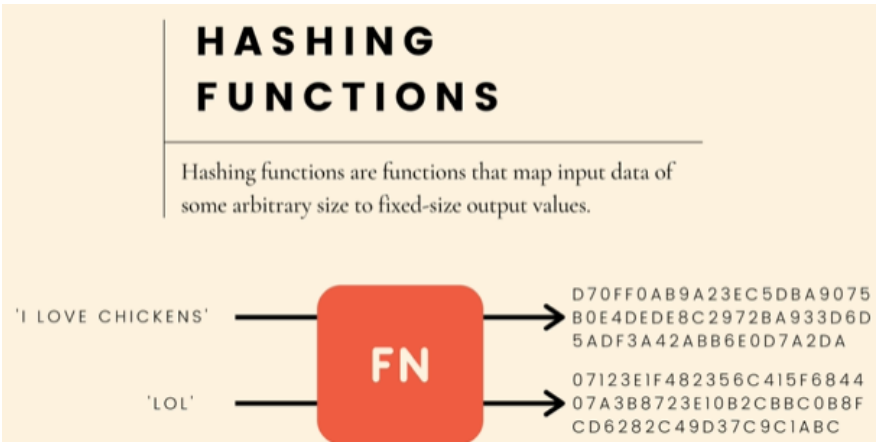


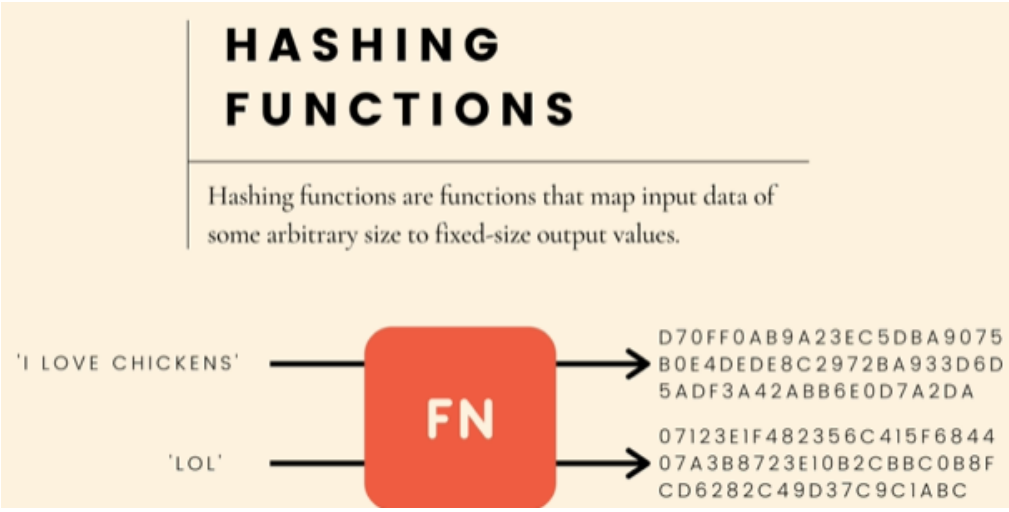
Git store full snapshot, it doesnt store diffs.

A Crash Course on Hashing Functions

```
commit 07027f93eaa05621da90f69e45d0787e5aaaf87d
```

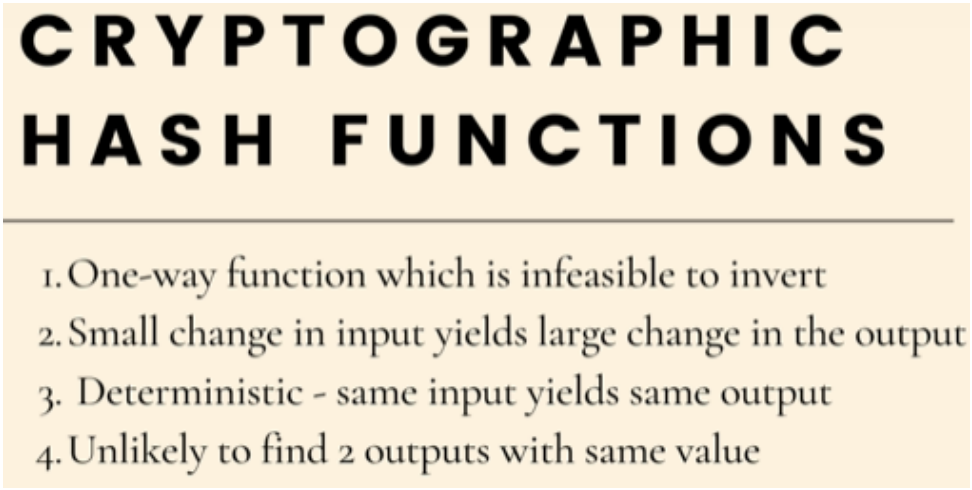
Hexadecimal character, exactly 40 digits long.





Hashing functions are family of functions that have one goal. They take input of arbitary size and spit out output of fixed size.

The input can be a sentence,3 characters, files or single number.



Cryptographic hash functions which is a subset of hash functions, they do the same thing, they map the data of arbitary size to an output of fixed size.

Use: info security, digital signatures, cryptocurrency , git.

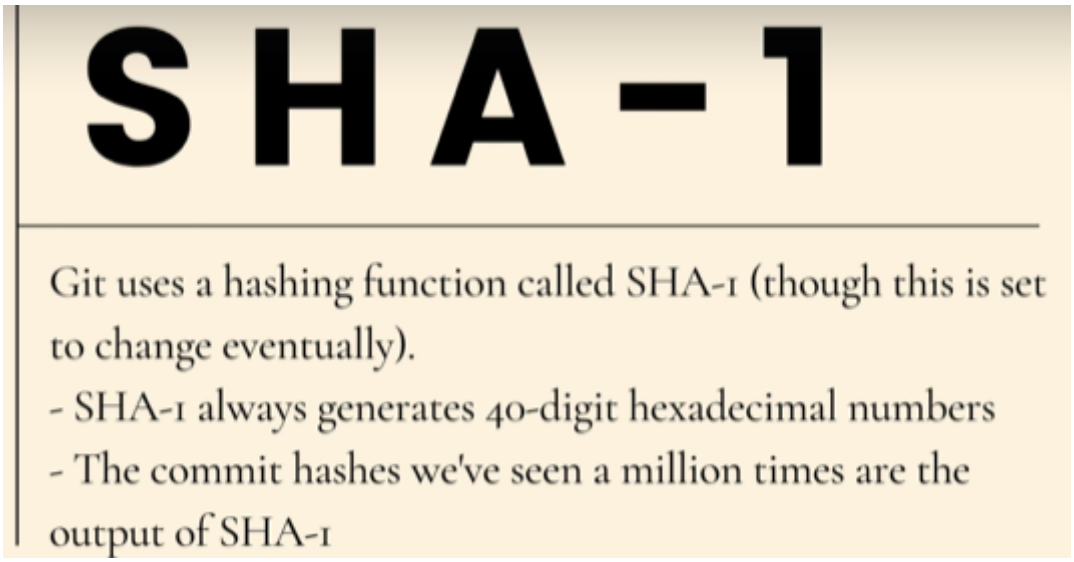
Deterministic, that “I love chickens” hash should always be the same. There should be consistency.

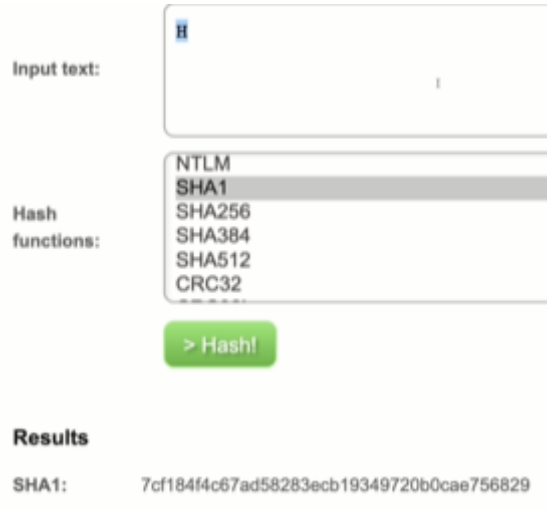
It should be a one way function, that is by knowing the output you cant be able to infer,guess or reverse engineer the output. So if i were to implement a password login authentication for a user on a website. We use hashing func to store hash

output of a password. So if somebody get access to our database, they see the password hashes.

Also a small change in input yields large change in the output.

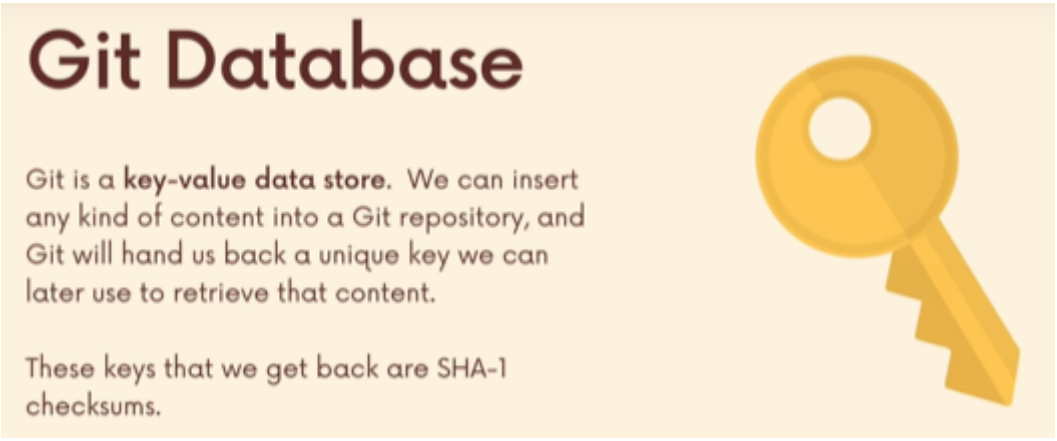
Unlikely to find 2 outputs with the same value.





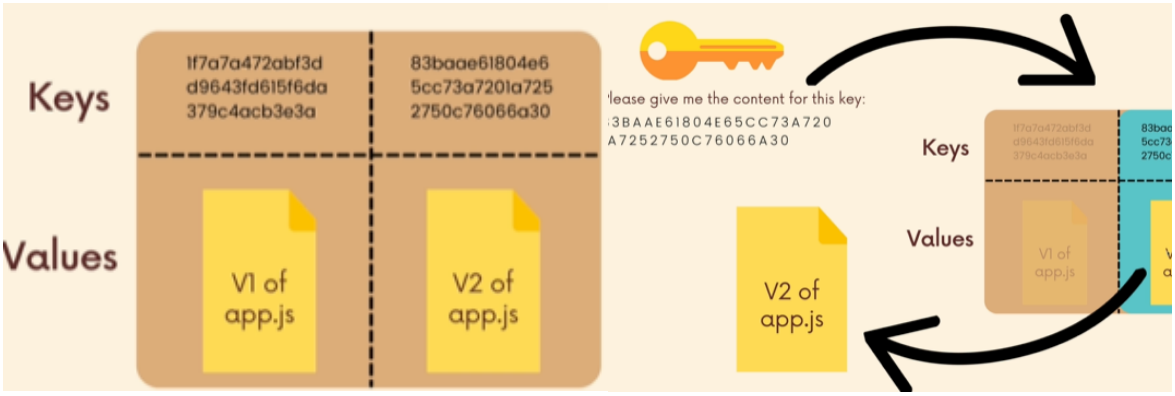
Git As A Key-Value Datastore

Git use SHA-1 for above.

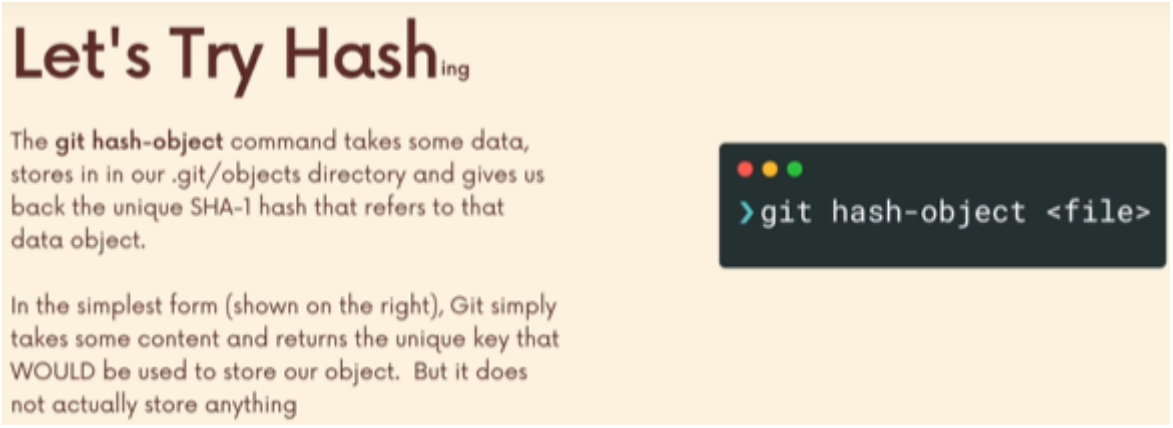


Outside dmart, they take ur stuff and give u a key or token.

Git is gonna give us a unique key which we gonna use to retrieve the content.



Hashing with Git Hash-Objects



It takes content from a file and it returns a unique SHA-1 hash that would be use to store our objects.

When we store something, git will create an entry in objects directory.


```
GitObjects › echo "hi" | git hash-object --stdin
45b983be36b73c0788dc9cbcb76cbb80fc7bb057
GitObjects › echo "hi" | git hash-object --stdin
45b983be36b73c0788dc9cbcb76cbb80fc7bb057
```

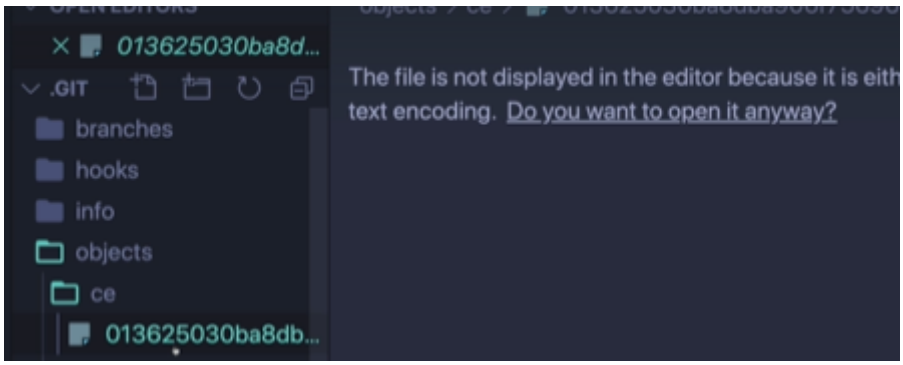
Just like i can run **git init** anywhere i can also run **git hash**.

```
- objects
  ├── 03
  │   └── 5dc548450f2a2b6a0f0eaf7bce9e138069c41e
  ├── 07
  │   └── 5b5312afe6c6f35f80810c848f9cd9ce78f2ac
  ├── 08
  │   └── ee35effe3e958221ea141c3c6690cc57263bfc
  ├── 0b
  │   └── d1260c1aa05d3058b592e289b0693b76326182
```

```
GitObjects › echo "hi" | git hash-object --stdin
45b983be36b73c0788dc9cbcb76cbb80fc7bb057
GitObjects › echo "hi" | git hash-object --stdin
45b983be36b73c0788dc9cbcb76cbb80fc7bb057
```

Note: Objects isnt changing at all, its just saying hello git, if u were to store anything what will be the hash.


```
GitObjects › echo "hello" | git hash-object --stdin -w
ce013625030ba8dba906f756967f9e9ca394464a
GitObjects ›
```




```
GitObjects › echo "goodbye" | git hash-object --stdin -w
dd7e1c6f0fefe118f0b63d9f10908c460aa317a6
```

Retrieving data with git Cat-File

Let's Try Hashing

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

Now that we have data stored in our Git object database, we can try retrieving it using the git cat-file command.

The -p option tells Git to pretty print the contents of the object based on its type.

```
GitObjects ❯ git cat-file -p ce013625030ba8dba906f756967f9e9ca394464a
hello
```

```
GitObjects ❯ git cat-file -p dd7e1
goodbye
```

```
GitObjects ❯ git hash-object dogs.txt -w
39e2751150ccb0f59ec1619021f66551f42138ec
```

```
GitObjects ❯ git hash-object dogs.txt
fd9150c2a3bb4081a41114065f42a1cd98d2124c
GitObjects ❯ git hash-object dogs.txt -w
fd9150c2a3bb4081a41114065f42a1cd98d2124c
```

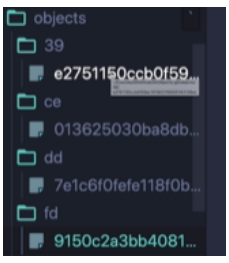
```
GitObjects ❯ git cat-file fd9150c2a3bb4081a41114065f42a1cd98d2124c -p
Rusty
Wyatt
Cheyenne
Sirius%
```

```
GitObjects ❯ git cat-file -p 39e2751
Rusty
Wyatt%
```

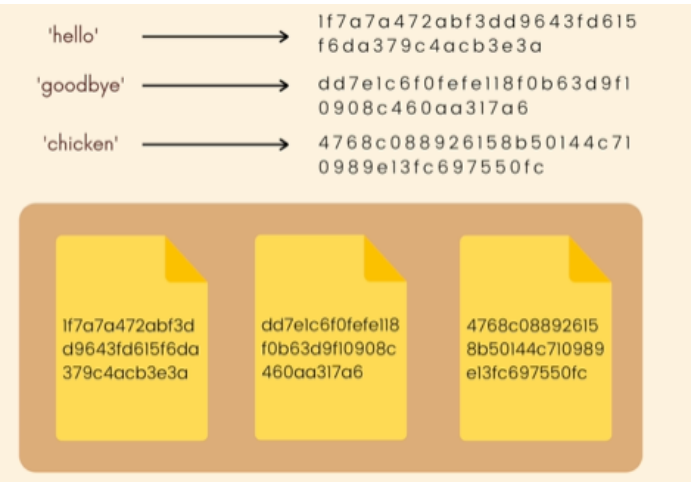
FYI, the dog.txt file was never committed.

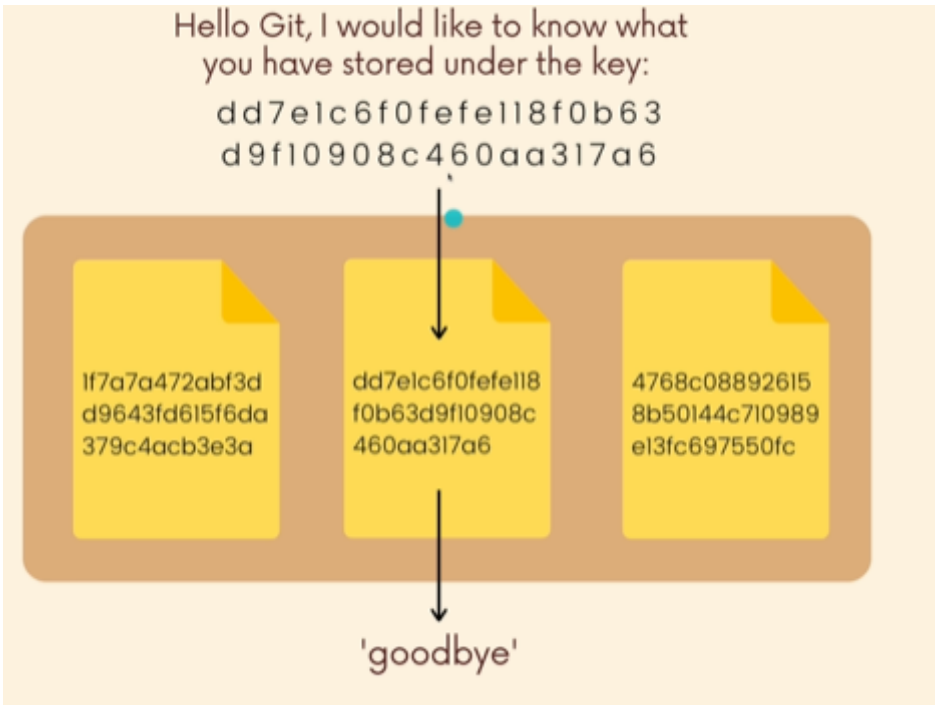
We see git store two different version of file.

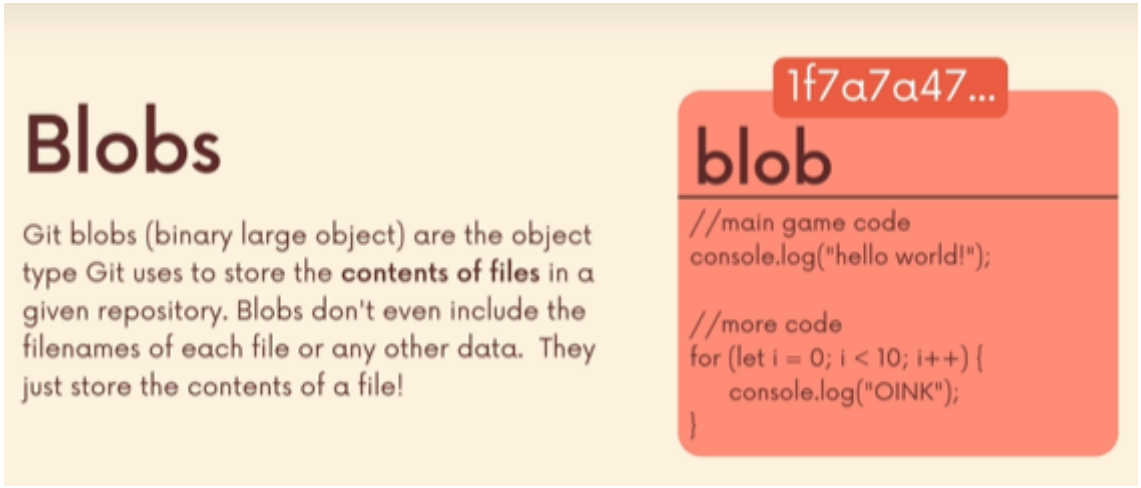
```
GitObjects ❯ git cat-file -p fd9150c2a > dogs.txt
GitObjects ❯
```

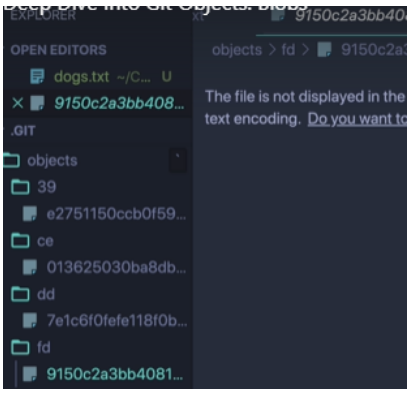


Deep dive Into Git Objects: Books

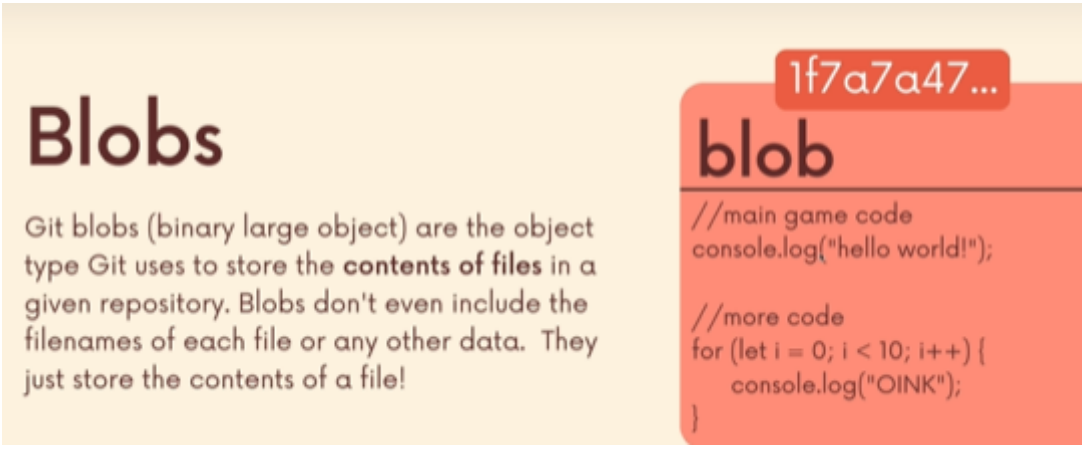








They all are blobs.



Git stores the content of a file, seperate from the main. Each blob get its own hash.

Blobs is the building block, the way we use git of different files, different contents ,different versions, different commits.

Blobs does get its own hash, its not a commit hash. It looks like commit hash cuz similar Hash func outputs.

Deep dive Into Git Objects: Trees

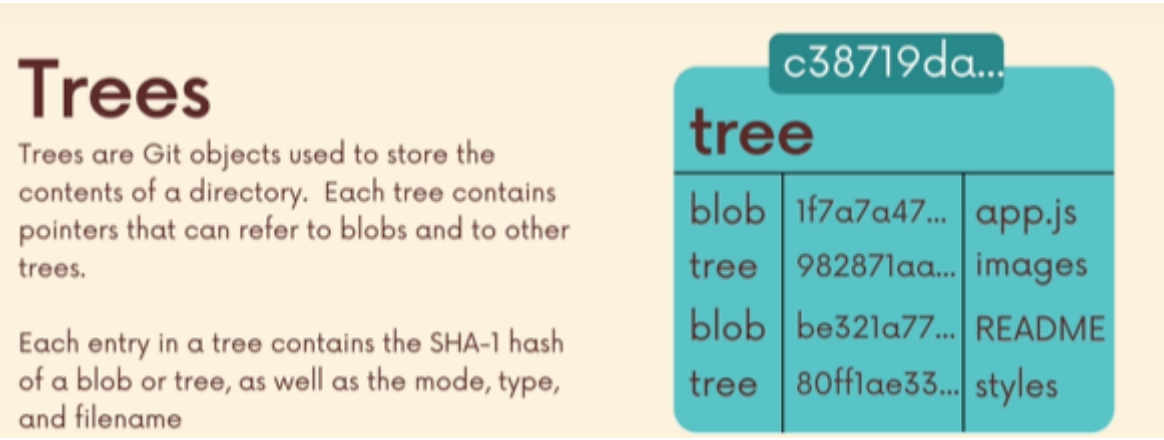
In a large project with 100 files, with nested folders. If i tell git to checkout this branch or commit. It can change all of those files,folders ; it can delete some in my working directory. It might add whole bunch of stuff.

How is it able to keep track of that structure.

Its not just the insides of the files, its the relationships between the files. Its the structure of directories.

Blobs store the contents of the files.

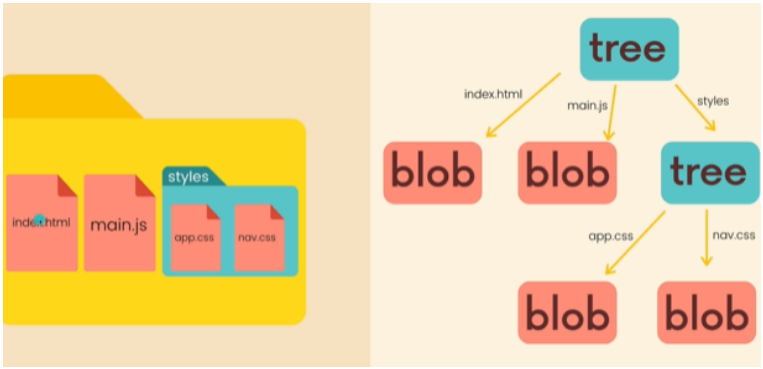
Trees stores the content of the directories.

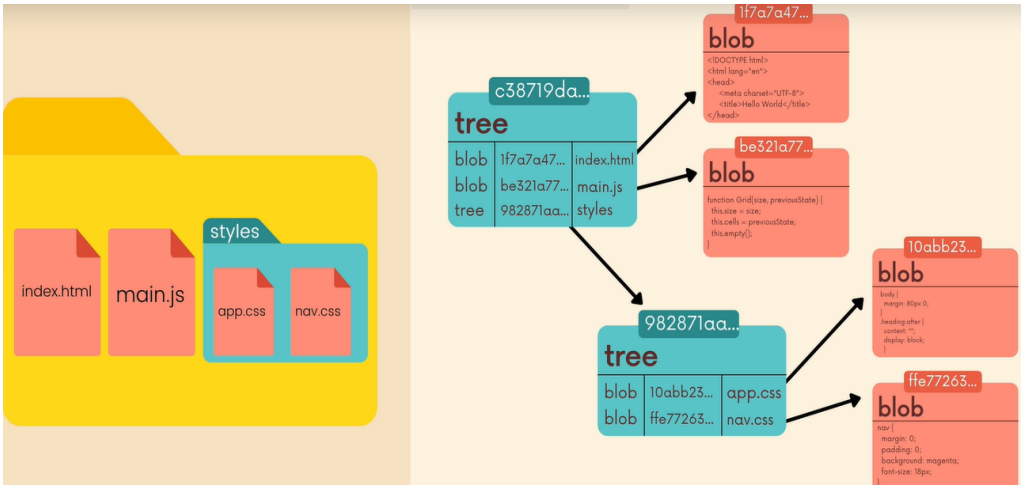



```
        └── 9dc115bb6286
    ├── ff
    │   └── 051eb0abf15d
    ├── info
    └── pack
└── refs
    ├── heads
    │   ├── Sahil
    │   ├── kazim
    │   ├── master
    │   └── sania
```

Above we have references to blob and trees.

Remember: The BLOB only stores the content referenced by a tree using some hash based on its contents. The tree maps file-names/folder to hashes.





In an existing repository that has history we can view trees.

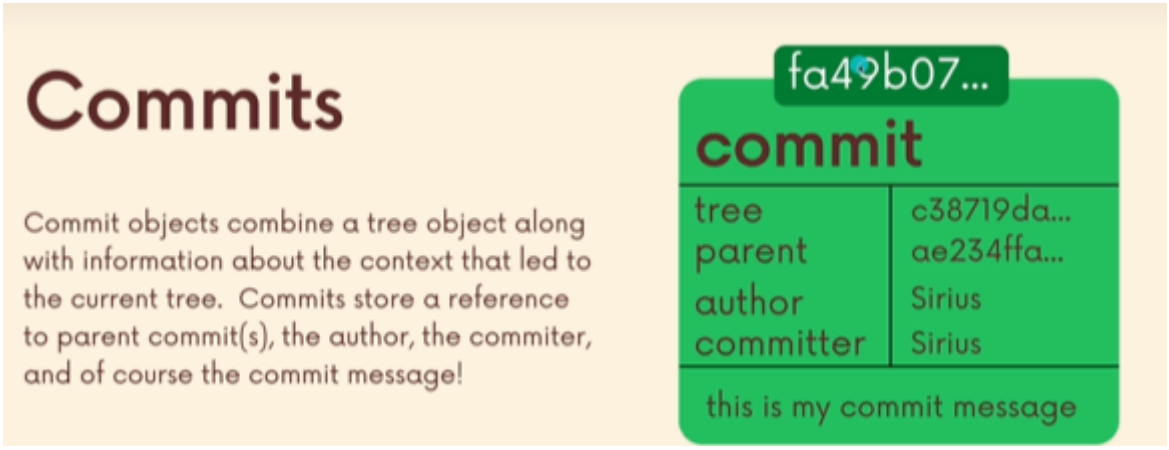
Master^{tree} is a particular way to refer to the tree object that is pointed to by the tip of our master branch.

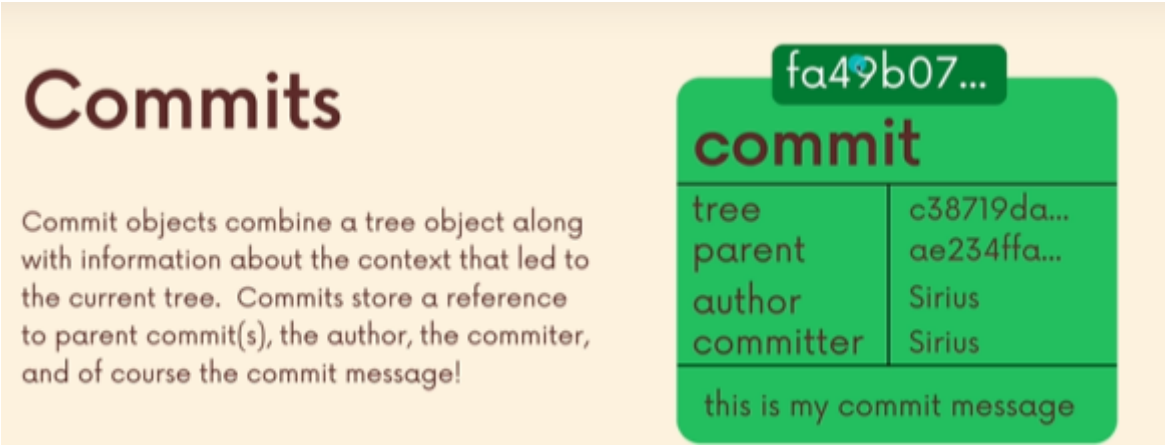
```
react › git cat-file -p master^{tree}                          master
040000 tree 71b2334fd6fd0355ecff70d313e073eae15b7dbb    .circleci
040000 tree 7699ec91f49741f8b71770bf778f85380d645f94    .codesandbox
100644 blob 07552cfff88bafaf4d207e6255394bc6d6215302    .editorconfig
100644 blob 2af8176bc3e122a83c45af585c5e8e27edf6f30f    .eslintignore
100644 blob a456964bdbdb0324bedb013c4cfe343a1a370d37    .eslintrc.js
100644 blob 176a458f94e0ea5272ce67c36bf30b6be9caf623    .gitattribute
100644 blob 1ea0baeb3e3b0fc01e844f5df3b55e387a91821a    README.md
100644 blob 655dfeaec0e67a9c448bf08a5f32d1f73aaa9611    SECURITY.md
```

```
react › git cat-file -p 1ea0baeb3e3b0fc01e844f5df3b55e387a91821a
```

```
react › git cat-file -t 1ea0baeb3e3b0fc01e844f5df3b55e387a91821a
blob
react › git cat-file -t 5c40176f073130c43b2d40bc473b791ac94ffe4e
tree
```

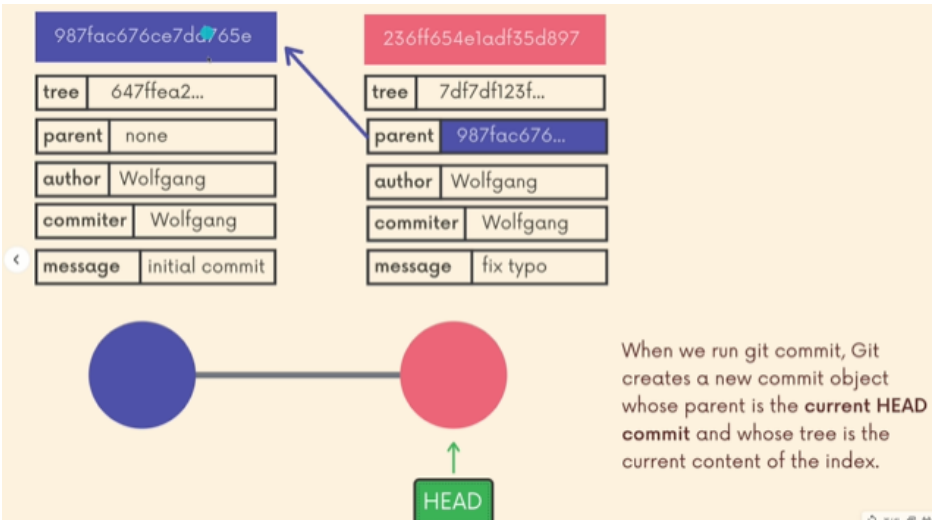
Deep Dive Into Git Objects: Commits





When we make a commit, git generates a new commit git object.

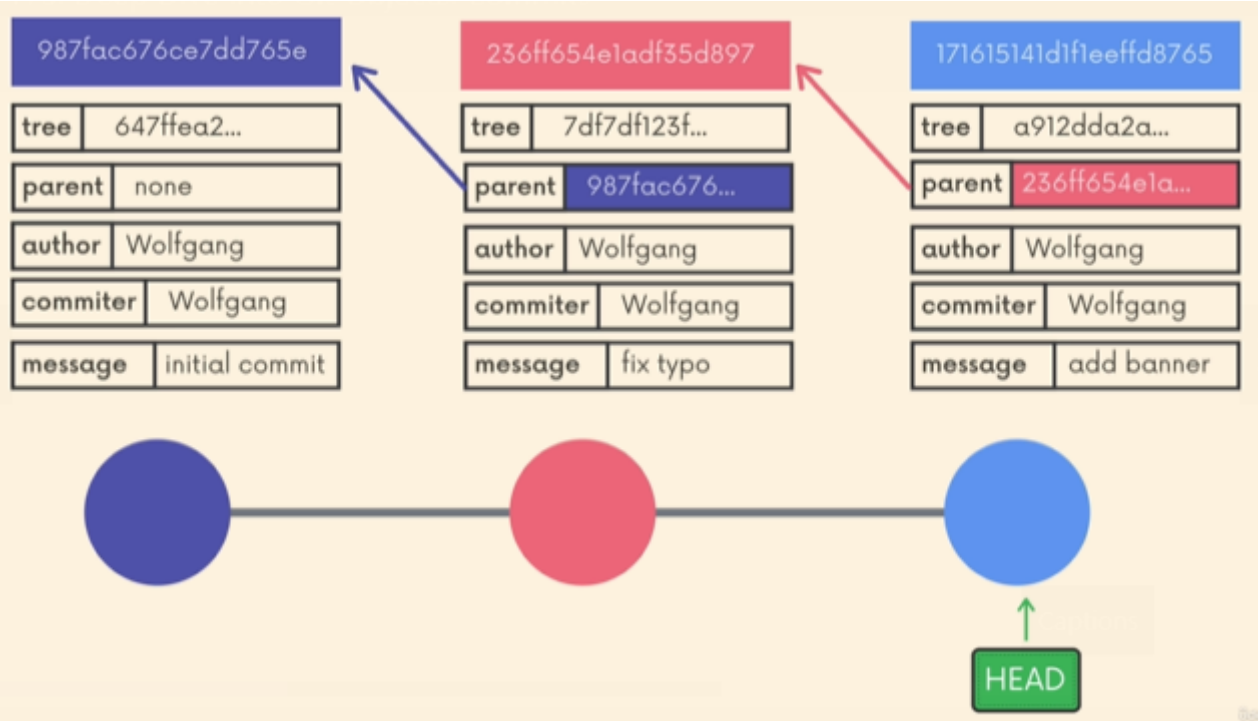
First two digits make a folder, inside that 38 digits remaining that is the name of the encrypted binary compressed files.



Also, the tree that it stores in this commit, the tree is the current content of this index, the staging area.

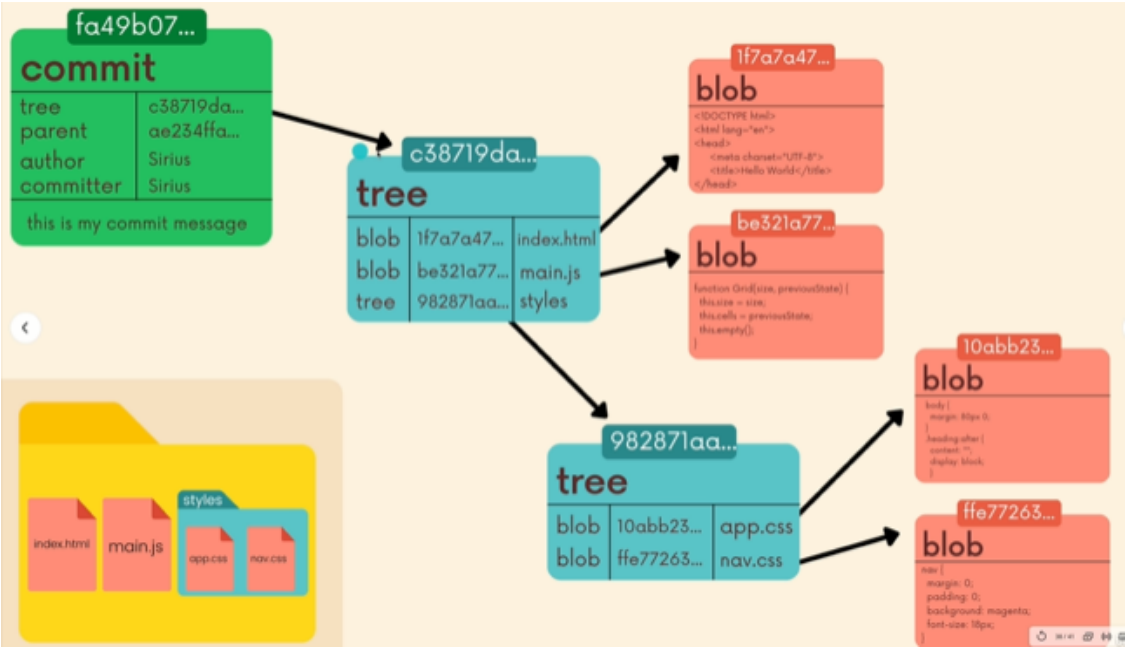
So there's an actual index file.

Basically, everything that we have worked on all along, the changes we have staged. When we run **git commit**, a tree is generated. That tree reflects the content of the index that is included in the commit.



Every commit is tied to a tree, that tree represents the structure of application i.e. other trees or blobs(which contains the content identified by a hash).

So if we checkout a commit, git is gonna use that tree and use it as the basis for our working directory.




```
GitObjects ❯ git commit -m "initial commit"
[master (root-commit) 79dd61f] initial commit
 1 file changed, 4 insertions(+)
 create mode 100644 dogs.txt
```

```
commit 79dd61fa59c1e3391443d89e14d3a7ebdcd2aabe (HEAD -> master)
Author: Colt Steele <5498438+Colt@users.noreply.github.com>
Date:   Wed Feb 17 14:00:45 2021 -0800

    initial commit
```

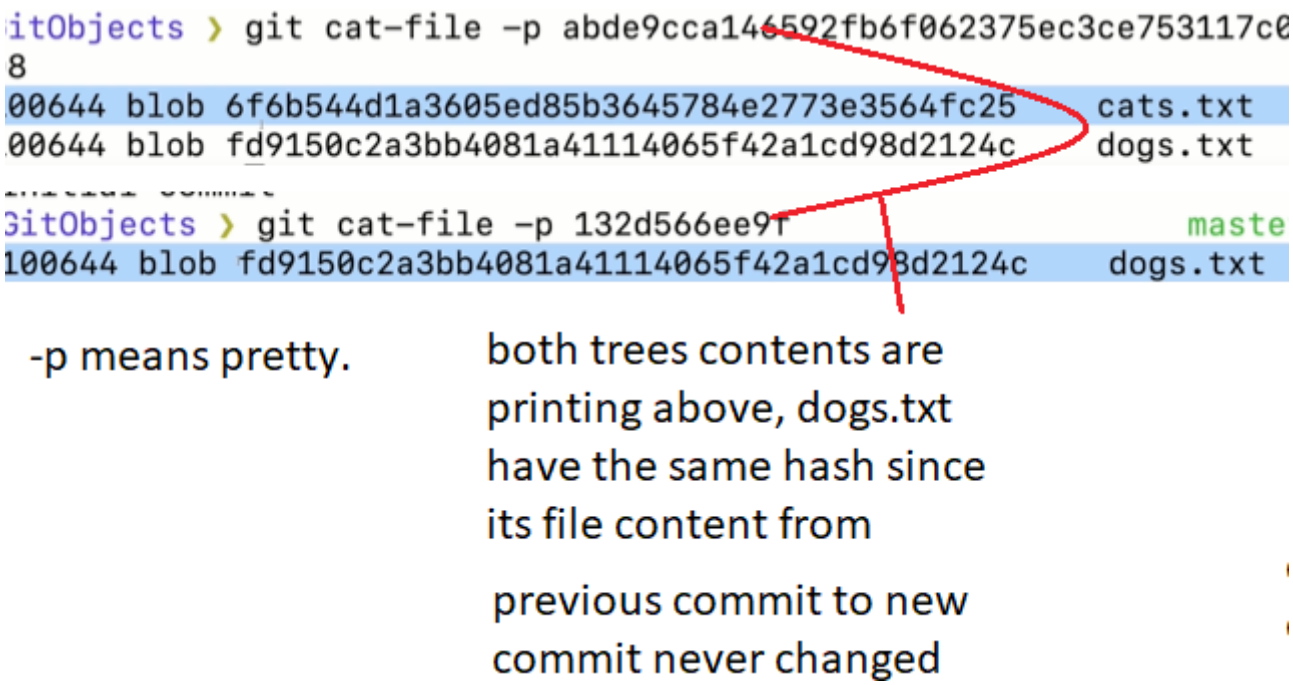
```
GitObjects ❯ git cat-file -t 79dd61fa59c1e3391443d89e14d3a7ebdcd2aa
be
```


```
GitObjects ❯ git cat-file -p 79dd61fa59c1e3391443d89e14d3a7ebdcd2aa
be
tree 132d566ee9f00789df1133d939f3779f639b9733
author Colt Steele <5498438+Colt@users.noreply.github.com> 16135992
45 -0800
committer Colt Steele <5498438+Colt@users.noreply.github.com> 16135
99245 -0800

initial commit
```

```
GitObjects ❯ git cat-file -t 132d566ee9f00789df1133d939f3779f639b97
33
tree
GitObjects ❯ git cat-file -p 132d566ee9f00789df1133d939f3779f639b97
33
100644 blob fd9150c2a3bb4081a41114065f42a1cd98d2124c    dogs.txt
```

```
GitObjects ❯ git commit -m "add cats file"
[master 51192c0] add cats file
 1 file changed, 1 insertion(+)
 create mode 100644 cats.txt
```

```
GitObjects ❯ git cat-file -p 51192c05c8e1e58a03fd4e3d8a3f55e9f388ac
7a
tree abde9cca146592fb6f062375ec3ce753117c0208
parent 79dd61fa59c1e3391443d89e14d3a7ebdcd2aabe
author Colt Steele <5498438+Colt@users.noreply.github.com> 16135993
91 -0800
committer Colt Steele <5498438+Colt@users.noreply.github.com> 16135
99391 -0800

add cats file
```



If i changed the contents of dogs.txt and commit, new hash will be created for dogs.txt.

Recap:

Every commit has a reference to a parent unless its the first commit.

Every commit has a tree thats a index that showcase the content of a directory.

i.e. references to other trees(files and folders) and blobs(file contents).

Everything gets hashed (blobs,trees,commits) and git uses this hashes to easily determine something is changed. Also uses the hashes as primary key to pick out file content.

When i switch branches, its gonna look at tip of the branch, whats that commit and whats the tree in that commit and restore the files in the working directory based on the state of the tree.

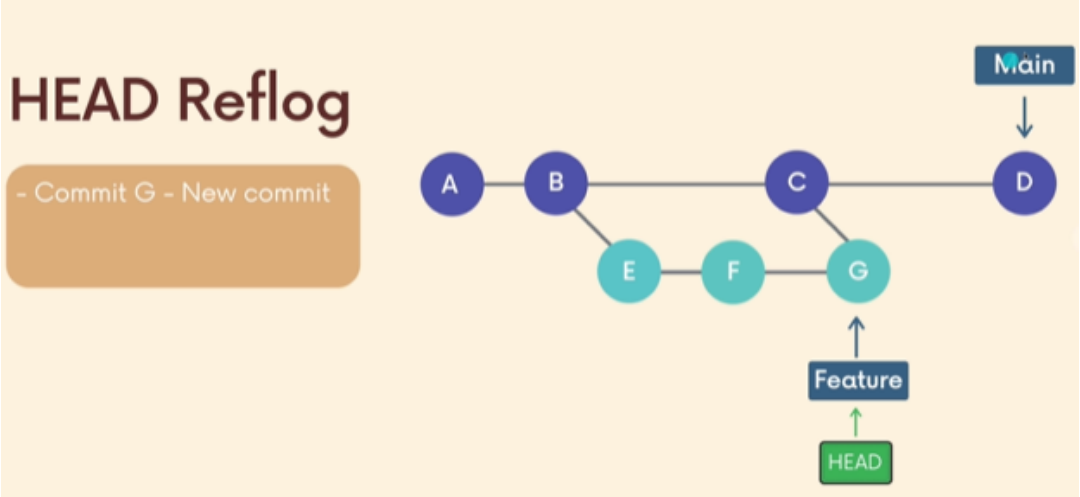
Annotated tag store a reference to a commit.

The Power of Reflogs, Retrieving Lost Work

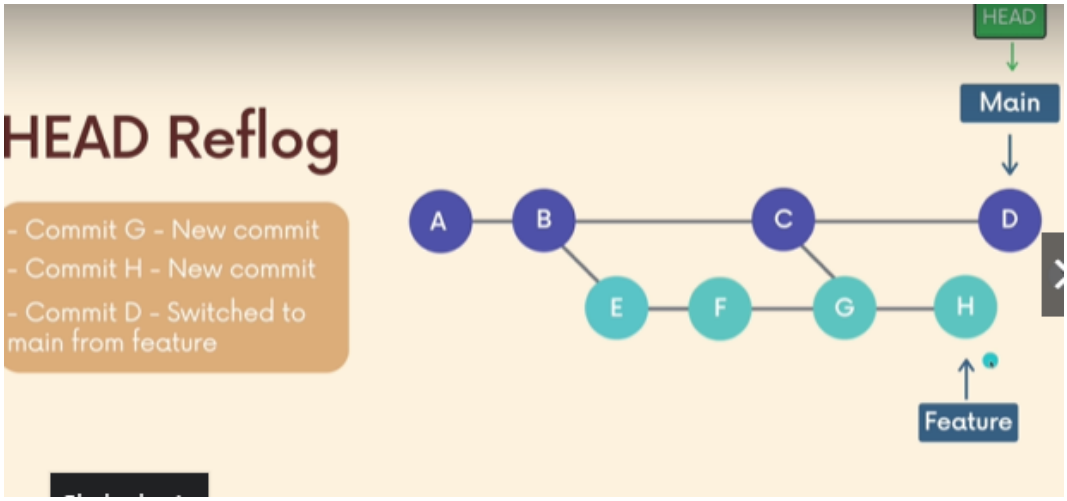
Reflogs is used when u screw something terribly or when i lost a commit or when i rebase when i shouldnt have rebase. When i want to undo something that seems undoable. With the help of **git reflog** often we can fix things, we can dig ourselves from pretty bad situations.

Introducing Reflogs

Reflog is short for reference logs.



Every branch has branch reference, that is what keeps the tip of the branch.




```
sahilsyed@DESKTOP-7VRCIK9:/mnt/c/Users/Dell/Desktop/Learning Git/kaju/.git$ ls
COMMIT_EDITMSG  HEAD  ORIG_HEAD  branches  config  description  hooks  index  in
 logs  objects  refs
```

```
c59d392617aed6732d76295a3a3a2317b403db38 3715203f52e4bef448e635ec94a1cac1f407bbfb
cher
3715203f52e4bef448e635ec94a1cac1f407bbfb 7d1e6b44b537d0d3b832291ba1494dbab737dd59
of my favourite and Sania
7d1e6b44b537d0d3b832291ba1494dbab737dd59 b6bcb5cdbd47f14faa911fc97b37d19c9d52fcfa
series
b6bcb5cdbd47f14faa911fc97b37d19c9d52fcfa 79f74d140f76f3525c5e6136079e9bc8fa802c3b
sania
79f74d140f76f3525c5e6136079e9bc8fa802c3b b6bcb5cdbd47f14faa911fc97b37d19c9d52fcfa
aster
kazim <kazimsyed9911@gmail.com> 1629886617 +0530      commit: middling evil in got,wit

kazim <kazimsyed9911@gmail.com> 1629886932 +0530      commit (merge): merge character

kazim <kazimsyed9911@gmail.com> 1629887669 +0530      commit: i also like marvel loki

kazim <kazimsyed9911@gmail.com> 1629887724 +0530      checkout: moving from master to

kazim <kazimsyed9911@gmail.com> 1629887741 +0530      checkout: moving from sania to m
```

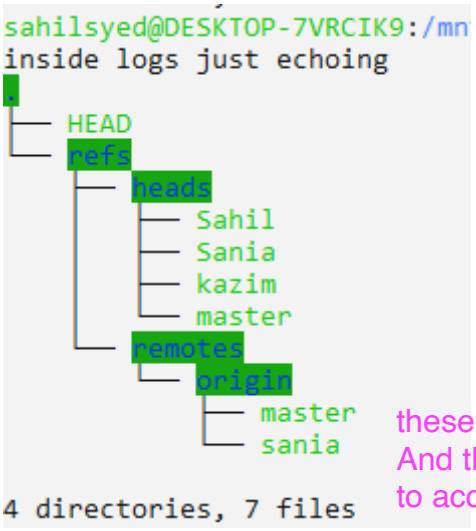
```
8ed9ea68fed4b11082d7b4cd5e31231531c97f23 68ed40038e4a6f4b93da91bcb09e2935f7db06ec chicken
little <chicken@gmail.com> 1613971302 -0800    commit: BLAH BLAH
68ed40038e4a6f4b93da91bcb09e2935f7db06ec 8ed9ea68fed4b11082d7b4cd5e31231531c97f23 chicken
little <chicken@gmail.com> 1613971443 -0800    reset: moving to HEAD~1
8ed9ea68fed4b11082d7b4cd5e31231531c97f23 d83634c58c1ce0bc563f625b24cda23355768044 chicken
little <chicken@gmail.com> 1613971671 -0800    commit: BLAH BLAH
```

```
8ed9ea68fed4b11082d7b4cd5e31231531c97f23
d83634c58c1ce0bc563f625b24cda23355768044
chicken little <chicken@gmail.com>
1613971671 -0800    commit: BLAH BLAH
d83634c58c1ce0bc563f625b24cda23355768044
143c628f5dcb63b02cdcb65bcb6f53b4c564668c
chicken little <chicken@gmail.com>
1614020062 -0800    checkout: moving from
master to turtle
```

```
Switched to branch 'turtle'
react ❯                          turtle
```

basically, log > refs> head, file tracks where the head is
so above, one way to change the head is
to
git switch turtle.

similarly



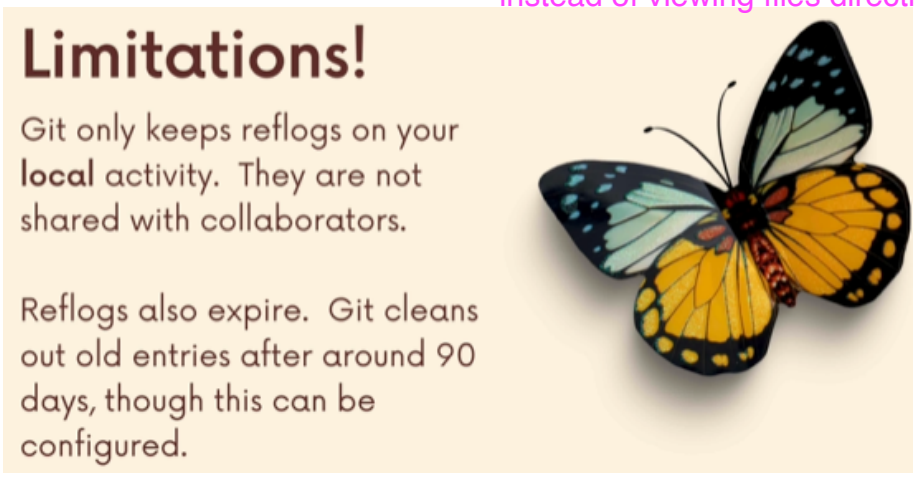
It keeps track of the tips of every branch.

every time i push up to origin/master.
there is a record of it.
its says update by push.

these logs exist. It's keeping logs that we can reference, later
And this can be a lifesaver at times if we make a mistake. If u need to access some commit hash that we no longer see.

The Limitations of Reflogs

instead of viewing files directly, we can use a git reflog cmd.



The Git Reflog Show command

Git Reflog

The git reflog command accepts subcommands **show**, **expire**, **delete**, and **exists**. Show is the only commonly used variant, and it is the default subcommand.

git reflog show will show the log of a specific reference (it defaults to HEAD).

For example, to view the logs for the tip of the main branch we could run **git reflog show main**.

```
> git reflog show HEAD
```

```
> git reflog
```

we can see for the first.
we are at where the head is pointing at master and to our knowledge and git knowledge we haven't deviated from the remote. which is true for what we know rn. also which is kinda telling since Head@{0} represents the most recent entry in the reflog.

Git reflog show HEAD/branchtip/remotebranch.

learn the language. if it was me i would have written. moved from there to master for the first reflog info.

```
react ) git log --oneline
react ) git reflog show HEAD
```

```
d83634c58 (HEAD -> master, colt/master) HEAD@{0}: checkout: moving
d01b9fc020131988d198f1c7799a33d725194 to master
295d01b9f HEAD@{1}: checkout: moving from turtle to 295d01b9f
143c628f5 (turtle) HEAD@{2}: checkout: moving from master to turtle
d83634c58 (HEAD -> master, colt/master) HEAD@{3}: commit: BLAH BLAH
8ed9ea68f HEAD@{4}: reset: moving to HEAD~1
68ed40038 HEAD@{5}: commit: BLAH BLAH
8ed9ea68f HEAD@{6}: checkout: moving from 78ec97d34aebce8a9970c5ee0
99d6e60 to master
78ec97d34 HEAD@{7}: checkout: moving from master to 78ec97d34
8ed9ea68f HEAD@{8}: checkout: moving from donkey to master
4898f7e78 (donkey) HEAD@{9}: checkout: moving from 6cdc35972d59d084
ee29787851f0e583 to donkey
6cdc35972 HEAD@{10}: checkout: moving from donkey to 6cdc35972
4898f7e78 (donkey) HEAD@{11}: checkout: moving from turtle to donke
143c628f5 (turtle) HEAD@{12}: checkout: moving from master to turtl
8ed9ea68f HEAD@{13}: commit: add hello file
01aafb33d HEAD@{14}: reset: moving to HEAD
01aafb33d HEAD@{15}: checkout: moving from master to master
```

colt open a new slate,not new but discarded the commit he has. It looks to me a plain reset ig.
the crazy part is, that he still have that hash. he removed. so u may lose hashes in ur git log, but they might be still there in reflogs. ofcoz the 90 days delivery.

We see a log ,similar to git log.

reflogs includes info that git logs doesnt.

It includes checkout to a branch.

git log doesn't care that u checkout a branch.
git log is just a log of commits
and its parents commit and so-on.

Also this going to be history of things, we could delete commits, we could rebase.

We still gonna see entries from when we did that.

```
143c628f5 (turtle) HEAD@{2}: checkout: moving f
d83634c58 (HEAD -> master, colt/master) HEAD@{3
8ed9ea68f HEAD@{4}: reset: moving to HEAD~1
68ed40038 HEAD@{5}: commit: BLAH BLAH
```

```
143c628f5 (turtle) HEAD@{2}: checkout: moving f
d83634c58 (HEAD -> master, colt/master) HEAD@{3
8ed9ea68f HEAD@{4}: reset: moving to HEAD~1
68ed40038 HEAD@{5}: commit: BLAH BLAH
```

Colt made mistakes and use **reset** to correct. He recorded the video couple of times.

You see, HE still have that hash.

But in **git log**, that hash is completely gone, Cuz he reset.

This history is only for 90 days.

Whereas **git log** will show me

```
d83634c58 (HEAD -> master, colt/master) BLAH BLAH
8ed9ea68f add hello file
01aafb33d add apple file
53e18e64c another commit yay
```

history from one commit to another commit.

```
d83634c58 (HEAD -> master, colt/master) HEAD@{0}: checkout: moving from 295
d01b9fc020131988d198f1c7799a33d725194 to master
295d01b9f HEAD@{1}: checkout: moving from turtle to 295d01b9f
143c628f5 (turtle) HEAD@{2}: checkout: moving from master to turtle
```

You can read, he moved from detached head to master.

```
d83634c58 (HEAD -> master, colt/master) HEAD@{0}: checkout
d01b9fc020131988d198f1c7799a33d725194 to master
295d01b9f HEAD@{1}: checkout: moving from turtle to 295d01
143c628f5 (turtle) HEAD@{2}: checkout: moving from master
d83634c58 (HEAD -> master, colt/master) HEAD@{3}: commit:
8ed9ea68f HEAD@{4}: reset: moving to HEAD~1
68ed40038 HEAD@{5}: commit: BLAH BLAH
8ed9ea68f HEAD@{6}: checkout: moving from 78ec97d34aebce8a
99d6e60 to master
78ec97d34 HEAD@{7}: checkout: moving from master to 78ec97
8ed9ea68f HEAD@{8}: checkout: moving from donkey to master
4898f7e78 (donkey) HEAD@{9}: checkout: moving from 6cdc359
ee29787851f0e583 to donkey
6cdc35972 HEAD@{10}: checkout: moving from donkey to 6cdc3
4898f7e78 (donkey) HEAD@{11}: checkout: moving from turtle
```

The most recent starts from Zero.

the illustration of this image is that we can reflogs to other branches/head/remote

the head@{0} represents the most recent thing in history.

```
react ❯ git switch donkey                                    master
Switched to branch 'donkey'
react ❯ git reflog show HEAD                                 donkey
react ❯ git reflog show donkey                               donkey
```

```
4898f7e78 (HEAD -> donkey) donkey@{0}: commit: add donkey file
ffe66ce99 (chicken) donkey@{1}: branch: Created from HEAD
```

sorry this ss doesnt make sense. the actual one says checking out from master to donkey

Passing Reflog References Around

Reflog References

We can access specific git refs is name{qualifier}. We can use this syntax to access specific ref pointers and can pass them to other commands including checkout, reset, and merge.

```
name@{qualifier}
```

Head@{2} means where head was two moves ago.

```
react ❯ git reflog show HEAD
react ❯ git reflog show HEAD@{10}
```

It shows me starting from 10, where the head was

```
8ed9ea68f HEAD@{10}: checkout: moving from donkey to master
4898f7e78 HEAD@{11}: checkout: moving from 6cdc35972d59d08451a529f1ee297878
51f0e583 to donkey
6cdc35972 HEAD@{12}: checkout: moving from donkey to 6cdc35972
4898f7e78 HEAD@{13}: checkout: moving from turtle to donkey
143c628f5 (turtle) HEAD@{14}: checkout: moving from master to turtle
8ed9ea68f HEAD@{15}: commit: add hello file
01aafb33d HEAD@{16}: reset: moving to HEAD
01aafb33d HEAD@{17}: checkout: moving from master to master
01aafb33d HEAD@{18}: commit: add apple file
53e18e64c HEAD@{19}: checkout: moving from turtle to master
```

```
react ❯ git checkout HEAD~2
```

~ means parent of head and its parent, two generation of parents ago.

Head~ is used for going backwards. i.e. two commits away from head

```
react ❯ git checkout HEAD@{2}
```

these may not be even commits that parents commits that we are going back to. we maybe on the same exact commit and we just switch branches cuz head@{2} says two moves ago in the ref log for head.
that might be changing branches or detached head.
who knows.

two entries ago from 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.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to fals
e

HEAD is now at d83634c58 BLAH BLAH
```

we can do fancier things, so instead of saying
2 moves ago, we can say lastweek or yesterday or an hour ago.

unlike the head reference, the head references updates when we switch branches, when i make commits, there's a bunch of things that changes head.
But when we do
#git reflog show donkey
Here this is going to show us the tip of the donkey branch and that only changes when donkey branch itself is updated.
so switching branches doesn't do anything.
But if i make a new commit in donkey branch.
we'll see a new entry/commit in git reflog donkey.

```
react ❯ git switch -
Previous HEAD position was d83634c58 BLAH BLAH
Switched to branch 'donkey'
```

```
7c13fefb3 (HEAD -> donkey) HEAD@{0}: checkout: moving from d83634c58c1ce0bc
563f625b24cda23355768044 to donkey
d83634c58 (colt/master, master) HEAD@{1}: checkout: moving from donkey to H
EAD@{2}
7c13fefb3 (HEAD -> donkey) HEAD@{2}: commit: add haha
4898f7e78 HEAD@{3}: checkout: moving from master to donkey
d83634c58 (colt/master, master) HEAD@{4}: checkout: moving from 295d01b9fc0
20131988d198f1c7799a33d725194 to master
295d01b9f HEAD@{5}: checkout: moving from turtle to 295d01b9f
```

it only show changes that matters.

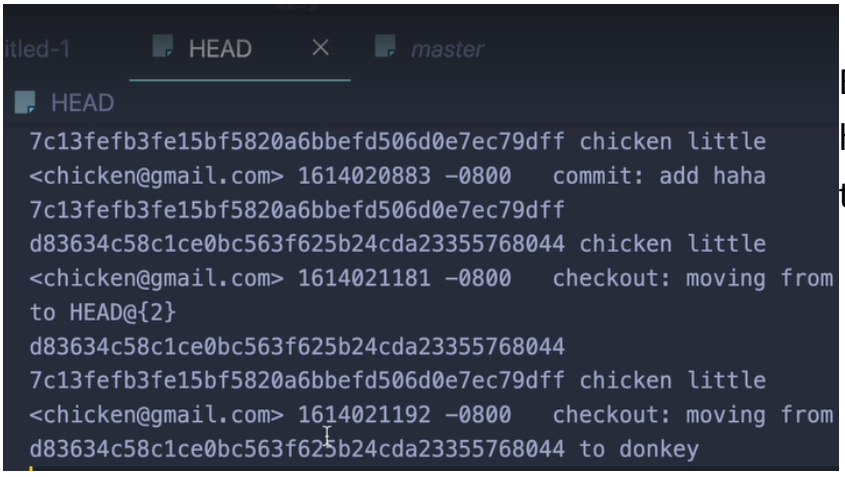
It wont show that i run

git log –oneline

we can ask for what changes on head now and 5 moves ago.

```
react ❯ git diff  HEAD@{0} HEAD@{5}
react ❯ git diff  HEAD@{0} HEAD@{2}
```

Time-Based Reflog Quantifiers



Every entry has a commit hash, the user who made it, timestamp.

We can see how the master branch look like yesterday, diff between today and yesterday work.

we can see what change between head today and head yesterday.

```
react ❯ git log --oneline
react ❯ git diff HEAD HEAD@{yesterday}
react ❯ git diff master HEAD@{yesterday}
react ❯ git reflog show HEAD@{2.days.ago}
react ❯ git reflog show HEAD@{two.days.ago}
react ❯ git reflog show HEAD@{one.minute.ago}
```

master

it can be 2 or two

Lets say uk things were working one week ago, so

```
react ❯ git reflog show HEAD@{one.month.ago}                    donkey
warning: log for 'HEAD' only goes back to Fri, 12 Feb 2021 14:06:21 -0800
react ❯ git checkout master@{1.week.ago}                        donkey
```

```
  git switch -

Turn off this advice by setting config vari
e

HEAD is now at 796f12225 add new feature
```

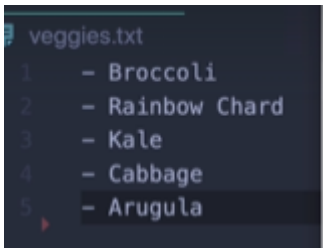
i dont have to find a commit hash, its gonna take me to the closest matching ref log entry.

Rescuing Lost Commits with Reflogs

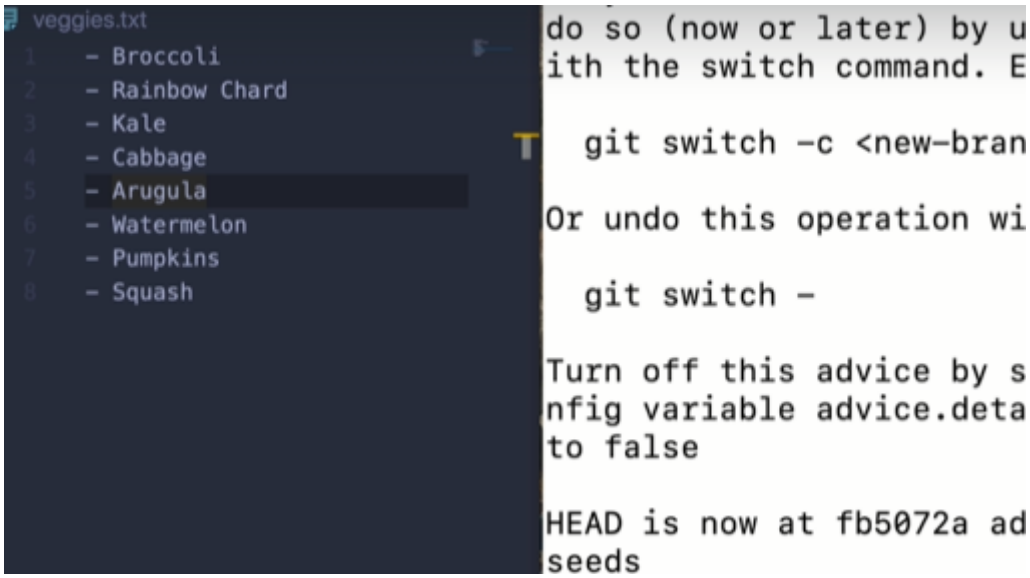
So, we can do hard reset and lose commits and still retrieve the data from reflogs.

```
        veggies.txt

nothing added to commit but untracked files p
o track)
Reflogs ❯ git add veggies.txt
Reflogs ❯ git commit -m "add veggies file"
        modified:   veggies.txt

no changes added to commit (use "git add" and/or '
Reflogs ❯ git commit -am "plant winter veggies"
Reflogs ❯ git commit -am "add more greens"
[master db727ba] add more greens
 1 file changed, 3 insertions(+), 1 deletion(-)
Reflogs ❯ git commit -am "add summer seeds"

Reflogs ❯ git log --oneline
Reflogs ❯ git reset --hard db727ba
db727ba (HEAD -> master) add more greens
afe0b32 plant winter veggies
1f29fe9 add veggies file
Reflogs ❯ git reflog show master
db727ba (HEAD -> master) master@{0}: reset: moving to db727ba
fb5072a master@{1}: commit: add summer seeds
db727ba (HEAD -> master) master@{2}: commit: add more greens
afe0b32 master@{3}: commit: plant winter veggies
1f29fe9 master@{4}: commit (initial): add veggies file
```

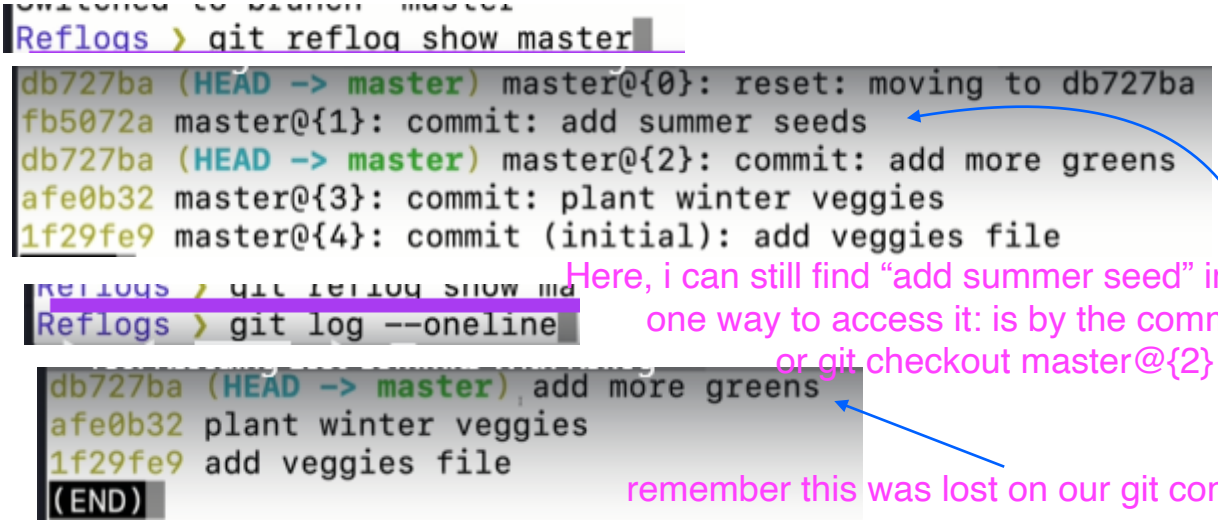
```
db727ba (HEAD -> master) master@{0}: reset: moving to db727ba
fb5072a master@{1}: commit: add summer seeds
db727ba (HEAD -> master) master@{2}: commit: add more greens
afe0b32 master@{3}: commit: plant winter veggies
1f29fe9 master@{4}: commit (initial): add veggies file
```




```
Reflogs ❯ git checkout fb5072a
```




```
Reflogs ❯ git switch -          HEAD
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  fb5072a add summer seeds

If you want to keep it by creating a new branch, this may be a good
time
to do so with:

 git branch <new-branch-name> fb5072a

Switched to branch 'master'
```

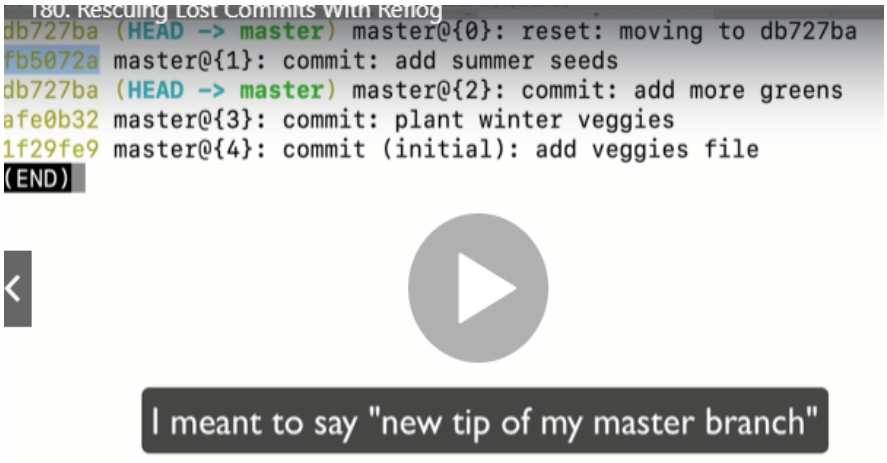


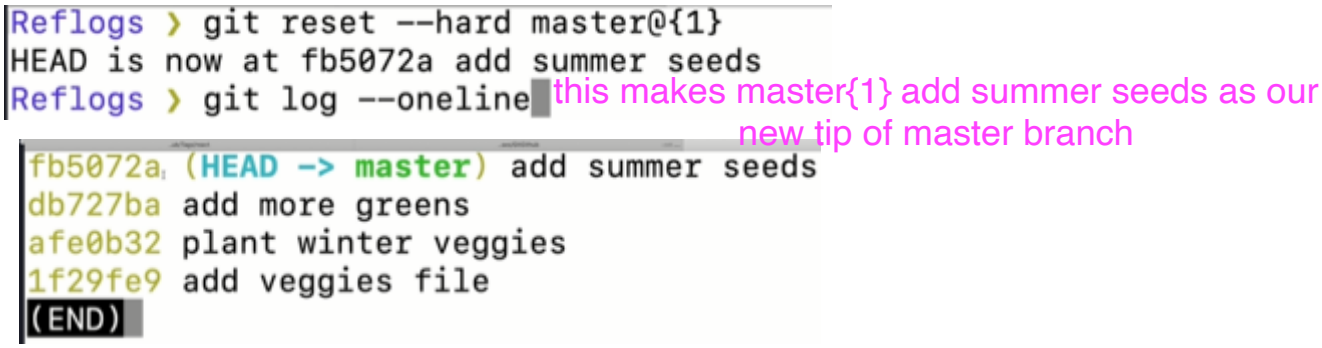
Even though "add summer seed" is long gone we can still find it reflogs.

Every so often git will go and remove commits that are dangling, that arent actually connected.

"add summer see" doesnt have a parent, child.

Since we have reflogs we can go and restore it.





We just undid a hard reset, keep in mind it only works on local changes.

Undoing A Rebase w/ Reflogs : - Its a miracle

Rebase rewrites commits and make new commits and those old ones are gone.

Thats what we used to know so far.

Heres the kicker.

```
Reflogs ❯ git switch -c flower
```

```
Reflogs ❯ git commit -am "add some
 zinnias"
[flowers ab60d1c] add some zinnias
 1 file changed, 2 insertions(+)
Reflogs ❯ git commit -am "add dahl
ias to flowers list"
[flowers 14fa331] add dahlias to f
lowers list
 1 file changed, 3 insertions(+),
1 deletion(-)
Reflogs ❯ git commit -am "add amar
anth to flowers"
[flowers 355885f] add amaranth to
flowers
 1 file changed, 3 insertions(+),
1 deletion(-)
Reflogs ❯                    flowers
```

```
- Salmon Zinnias
- Queen Lime Zinnias
- Old Rose Dahlias
- Dinnerplate Dahlias
- Coral Fountain Amaranth
- Hot Biscuits Amaranth
- Coral Reef Celosia
```

```
Reflogs ❯ git rebase -i HEAD~4
hint: Waiting for your editor to c
lose the file...
```

fixup
fixup
fixup

```
pick ab60d1c add some zinnias
pick 14fa331 add dahlias to flowers list
pick 355885f add amaranth to flowers
pick c47ecfc add celosia

# Rebase dbbb9cd..c47ecfc onto c47ecfc (4 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase
--continue')
```

```
Reflogs ❯ git log --oneline
Reflogs ❯ git rebase -i HEAD~4
hint: Waiting for your editor to c
Successfully rebased and updated r
efs/heads/flowers.
```

```
51a5012 (HEAD -> flowers) add som
 zinnias
dbbb9cd add flowers file
fb5072a (master) add summer seeds
db727ba add more greens
afe0b32 plant winter veggies
1f29fe9 add veggies file
```

I can repeat the process one more time.

```
Reflogs ❯ git rebase -i HEAD~2
hint: Waiting for your editor to c
lose the file...
```

here we are fixing up and then rewording the commit.

```
reword dbbb9cd add flowers
file
fixup 51a5012 add some
zinnias
```

```
COMMIT_EDITMSG ●
COMMIT_EDITMSG
add list of flower seeds
```

```
ab9fef3 (HEAD -> flowers) add list
 of flower seeds
fb5072a (master) add summer seeds
db727ba add more greens
afe0b32 plant winter veggies
1f29fe9 add veggies file
```

then i realize i dont wanna do any of this.
so what should i do.
how will i turnover a rebase.

```
Reflogs ❯ git reflog show flowers
```

```
ab9fef3 (HEAD -> flowers) flowers@{0}: rebase (finish): refs/heads/f
lowers onto fb5072a95f8d1dd8c32e77ca8917b78021dec57f
51a5012 flowers@{1}: rebase (finish): refs/heads/flowers onto dbbb9c
d27f3470584fe8aa227c5ecc41c1798fba
c47ecfc flowers@{2}: commit: add celosia
355885f flowers@{3}: commit: add amaranth to flowers
14fa331 flowers@{4}: commit: add dahlias to flowers list
ab60d1c flowers@{5}: commit: add some zinnias
dbbb9cd flowers@{6}: commit: add flowers file
fb5072a (master) flowers@{7}: branch: Created from HEAD
```

Lets say i want to go to the below highlighted commit

So i can have all of the prior commits.

I can use the hash or flowers@{2}.

The commit is still stored in .git directory in the objects directory. It wont be there forever. Git goes through it periodically and it cleans up some of this as garbage collection. If it notices objects that are not reachable.

```
Reflogs ❯ git reset --hard c47ecfc
HEAD is now at c47ecfc add celosia
```

the end here.

```
c47ecfc (HEAD -> flowers) add celosia
355885f add amaranth to flowers
14fa331 add dahlias to flowers list
ab60d1c add some zinnias
dbbb9cd add flowers file
fb5072a (master) add summer seeds
db727ba add more greens
afe0b32 plant winter veggies
1f29fe9 add veggies file
```

The reason this work, every commit inside the file has its parent commit.

I CAN UNDO THE UNDOING BY GOING TO REFLOGS

```
c47ecfc (HEAD -> flowers) flowers@{0}: reset: moving to c47ecfc
ab9fef3 flowers@{1}: rebase (finish): refs/heads/flowers onto fb5072
a95f8d1dd8c32e77ca8917b78021dec57f
51a5012 flowers@{2}: rebase (finish): refs/heads/flowers onto dbbb9c
d27f3470584fe8aa227c5ecc41c1798fba
c47ecfc (HEAD -> flowers) flowers@{3}: commit: add celosia
355885f flowers@{4}: commit: add amaranth to flowers
14fa331 flowers@{5}: commit: add dahlias to flowers list
ab60d1c flowers@{6}: commit: add some zinnias
dbbb9cd flowers@{7}: commit: add flowers file
fb5072a (master) flowers@{8}: branch: Created from HEAD
```

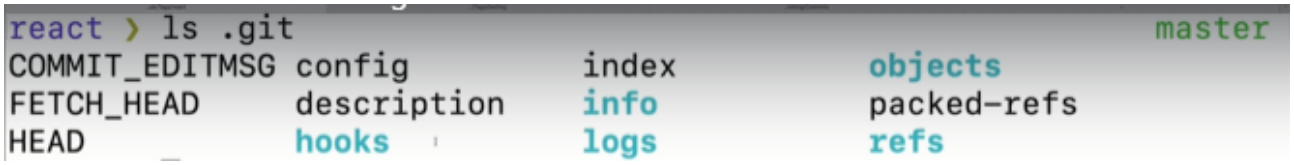
Writing Custom git aliases

How to define aliases and put into global config file.

The Global Git Config File

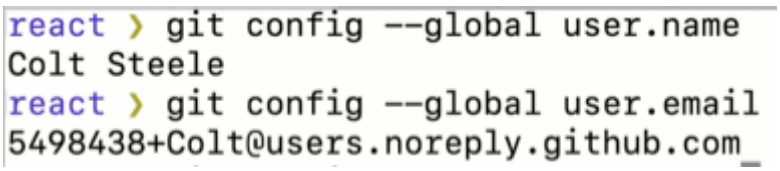
There is a local config file in every single git repo.

So setting in there, apply to one- repository.



Here, we can config settings that apply to all repository.

There is a one level above that, that is system wide. If u have multiple user accounts on one machine. That system wide will apply to all users.



Global Git Config

Git looks for the global config file at either ~/.gitconfig or ~/.config/git/config. Any configuration variables that we change in the file will be applied across all Git repos.

We can also alter configuration variables from the command line if preferred.

```
react ❯ cat ~/.gitconfig                                                    master
[filter "lfs"]
        required = true
        clean = git-lfs clean -- %f
        smudge = git-lfs smudge -- %f
        process = git-lfs filter-process
[user]
        name = Colt Steele
        email = 5498438+Colt@users.noreply.github.com
[core]
        excludesfile = /Users/coltsteele/.gitignore_global
        editor = code --wait
[difftool "sourcetree"]
        cmd = opendiff \"$LOCAL\" \"$REMOTE\"
        path =
[mergetool "sourcetree"]
        cmd = /Applications/Sourcetree.app/Contents/Resources/opend
iff-w.sh \"$LOCAL\" \"$REMOTE\" -ancestor \"$BASE\" -merge \"$MERGE
```

```
[user]
    name = Colt Steele
    email = 5498438+Colt@users.noreply.github.com
[core]
    excludesfile = /Users/coltsteele/.gitignore_global
    editor = code --wait
[difftool "sourcetree"]
    cmd = opendiff \"$LOCAL\" \"$REMOTE\"
    path =
[mergetool "sourcetree"]
    cmd = /Applications/Sourcetree.app/Contents/Resources/opendiff-w.sh
    \"$LOCAL\" \"$REMOTE\" -ancestor \"$BASE\" -merge \"$MERGED\"
    trustExitCode = true
```

This did the same exact thing if i do this.

```
react ❯ git config --global user.name "sadasd"
```

Writing our First Git alias

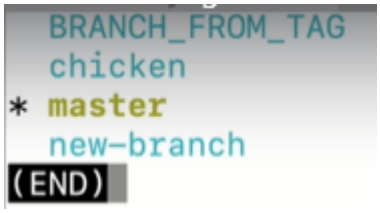
Only status.

We still need to run git s.

```
react ❯ git s                                                    master
On branch master
Your branch and 'origin/master' have diverged,
and have 3 and 3 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean
```

Setting Aliases From the Command Line

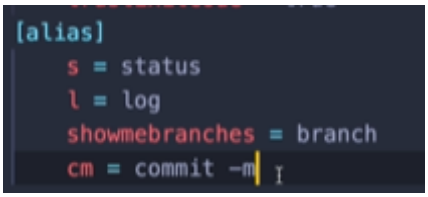
```
react ❯ git config --global alias.showmebranches branch      master
react ❯ git showmebranches                                    master
```



equivalent to git branch

here git is taking arguments in the
home ~
>
gitconfig
>
under [alias]
cm=commit -m

Aliases with Arguments




```
react ) git cm "my new commit!"
dquote> "
[master 295d01b9f] my new commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 new.js
```

```
commit 295d01b9fc0201319
Author: chicken little <
Date:   Wed Feb 17 21:29

    my new commit
```

Exploring Existing useful Aliases

Links:

Ultimate Git Alias Setup

must have git alias advance example

Git Alias Github Repo

```
[alias]
    s = status
    l = log
    showmebranches = branch
    cm = commit -m
    a = add
    ls = log --pretty=format:"%C(yellow)%h%Cred%d\\ %Creset%s%Cblue\\
    [%cn]" --decorate
    ll = log --pretty=format:"%C(yellow)%h%Cred%d\\ %Creset%s%Cblue\\
    [%cn]" --decorate --numstat
```

Rename [branch] to done-[branch]

In some of my workflows I wanted to quickly rename branches prepending `done-` to their names. Here is the alias that came out of that workflow:

```
done = "!f() { git branch | grep "$1" | cut -c 3- | grep -v done | xargs -I{} git branc
```

Last one is pretty good.