

GIT NOTES

by kante-srikanth

Note: This document content is gathered from various resources available in internet and solely responsible for learning purpose.

We can configure git at different levels:

| | |
|--------|---|
| System | # applicable to all users |
| Global | # applicable to current logged in user [all repos] |
| Local | # applicable to particular repo |

Configuration

Below config settings are stored in .gitconfig text file

```
git config --global user.name "kante-srikanth"
git config --global user.email kantesrikanth@gmail.com
git config --global core.editor "code --wait" # set default code editor git config --
global -e # opens config file i.e. .gitconfig
```

Help

```
git commandname --help # ex: git config -h / git config --help git help commandname
```

Git project Initialization

```
git init # initializes git in our project
```

Staging files

```
git add -A # stages new, deleted, modified files in entire work tree. Even if ur at folder level.
git add -A folder/ # stages same as above but only in folder/ path.
git add # default behavior as above commands.
git add -u # stages deleted, modified but not untracked / new files in entire tree.
git add -u folder/ # specific to folder path
git add . # stages the current directory and all its content
git add *.js # stages with a pattern
git add file1.js # stages a single file
git add file1.js file2.js # stages multiple files at a time
```

Viewing the status

```
git status # gives complete status
git status -s # gives short status
```

Committing the staging files

```
git commit -m "Message" # commits with a one-line message git commit
# opens the default editor to type a long message git commit -
am "Message" # skips the staging area
```

Unstaging files / (undoing git add)

```
git reset
# unstages the files; brings all the staged files back to working directory state. [used in git old versions] git
restore --staged file.js
```

Copies last version of file.js from repo to index i.e restores the staging area [same as above but new command]

Removing files `git rm file1.js` # Removes from working directory and staging area [internally does `git add .`]
`git rm --cached file1.js` # Removes from staging area only [we need to explicitly perform `git add .`]

Discarding & Restoring

| | |
|---|---|
| <code>git restore file.js</code> | # copies file.js from index to working directory |
| <code>git restore file1.js file2.js</code> | # restores multiple files in working directory |
| <code>git restore</code> | # discards all local changes (except untracked files) |
| <code>git clean -fd</code> | # removes all untracked files |
| <code>git restore --source=HEAD~1 filename</code> | # restore file from previous commit history i.e HEAD -1 |

Renaming or moving files:

| | |
|--|---|
| <code>git mv file1.js file1.txt</code> | # renames the file name or moves the file |
|--|---|

Difference

| | |
|--|--|
| <code>git diff</code> | # shows only when there are unstaged changes |
| <code>git diff --staged</code> | # shows staged changes |
| <code>git diff --cached</code> | # same as the above |
| <code>git diff HEAD~2 HEAD</code> | # shows the changes between two commits |
| <code>git diff HEAD~2 HEAD file.txt</code> | # shows changes for file.txt only |

Commit History && Browsing:

| | |
|---------------------------------------|--|
| <code>git log</code> | # shows history of commits, without filenames |
| <code>git log --oneline</code> | # shows one line for history of commits |
| <code>git log --reverse</code> | # shows in reverse order |
| <code>git log --stat</code> | # shows commit history as well as files names of modified. |
| <code>git log --patch</code> | # shows the content changed (patches) |
| <code>git log file.txt</code> | # shows the commits that modified file.txt |
| <code>git log --stat file.txt</code> | # shows statistics (the number of changes) for file.txt |
| <code>git log --patch file.txt</code> | # shows the patches (changes) applied to file.txt |

Filtering the history

| | |
|--|----------------------------|
| <code>git log -3</code> | # shows the last 3 entries |
| <code>git log --author="kante-srikanth"</code> | |
| <code>git log --before="2020-08-17"</code> | |
| <code>git log --after="one week ago"</code> | |
| <code>git log hash1...hash2</code> | # range of commits |

Commit viewing

| | |
|---|---|
| <code>git show commitid</code> | # shows commit changes and history |
| <code>git show HEAD</code> | # shows commit changes and history of last commit |
| <code>git show HEAD~1</code> | # shows the last commit - 1 . |
| <code>git show commitid:fullpathtofilename</code> | # shows the exact content of the file |
| <code>git ls-tree commitid</code> | # shows all files changed in the given commit] |

Creating an alias

| | |
|---|--|
| <code>git config --global alias.lg "log --oneline"</code> | # create alias for command ex: <code>git lg</code> <code>git log --oneline</code> |
|---|--|

Checkout

| | |
|-----------------------|---|
| git checkout commitid | # checkout given commit, will be in detached HEAD state |
| git checkout master | # checkout master branch |
| git checkout filepath | # undo all the changes in working directory of the specified file |

Working with remote repo

| | |
|-------------------------|--|
| git clone repo_url | # clones remote repo to local |
| git fetch origin master | # fetches master from origin but don't merge |
| git fetch origin | # fetches all objects from origin |
| git fetch | # shortcut for "git fetch origin" git |
| pull | # fetch + merge |
| git push origin master | # pushes master to origin git push |
| | # shortcut for "git push origin master" |

Pushing & Pulling changes to repo

| | | |
|------------------------------------|--|-----|
| git add . | # add changes to staging area | |
| git commit -m "message" | # committing to local repo i.e create snapshot | git |
| push origin branchname or git push | # push local changes to remote repo | |
| git pull origin branchname | # pull latest changes from repo to local | |

Managing branches

| | |
|------------------------|--|
| git branch bugfix | # creates a new branch called bugfix |
| git checkout bugfix | # switches to the bugfix branch |
| git switch bugfix | # same as the above |
| git switch -C bugfix | # creates and switches |
| git checkout -b bugfix | # same as above |
| git branch -d bugfix | # deletes the bugfix branch |
| git branch | # this will list all local branches |
| git branch -a | # this will list all local and remote branches |

Comparing branches

| | |
|--------------------------|--|
| git log master bugfix | # lists the commits/changes in the bugfix branch not in master |
| git diff branch1 branch2 | # shows the summary of changes, shows <i>branch1 – branch2</i> |

Sharing branches

| | |
|---------------------------|--|
| git branch -r | # shows remote tracking branches |
| git branch -vv | # shows local & remote tracking branches |
| git push -u origin bugfix | # pushes bugfix to origin |
| git push -d origin bugfix | # removes bugfix from origin |

Managing remotes

| | |
|-----------------------------|--|
| git remote | # shows remote repos |
| git remote add upstream url | # adds a new remote called upstream <i>upstream == remote repo</i> |
| git remote rm upstream | # removes upstream |

Finding contributors:

| | |
|-----------------|--|
| git shortlog -h | # gives info about authors, no of commits etc. |
|-----------------|--|

Blame

git blame filename.ext

shows author of each line in terminal

Merge

git merge bugfix

merges the bugfix branch into the current branch

git merge --no-ff bugfix

creates a merge commit even if FastForward is possible

git merge --squash bugfix

performs a squash merge

git merge --abort

aborts the merge

git branch --merged

shows the merged branches

git branch --no-merged

shows the unmerged branches

Create local branch && pushing to origin

git branch branchname git checkout branchname

git checkout -b branchname

git push -u origin branchname

#Associates local branch with remote and push changes. so that others can also work on the same branch.

Steps for Merging branch to master

git checkout master # first checkout local master git pull origin master

then pull the changes git branch --merged # check

whether our branch is merged or not git merge branchname # this will

merge the branch into local master git push origin master # then push the local

master to repo master ..DONE

After merging the branch, Delete the local and remote branch

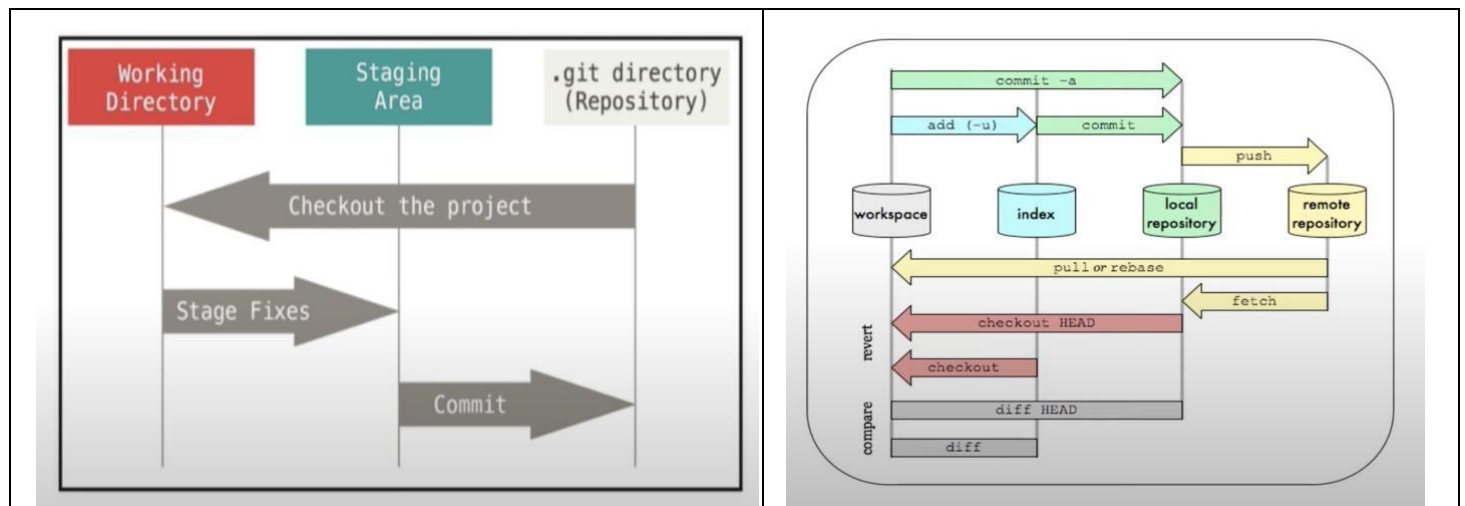
git branch --merged # check if our local branch is merged into local master git

branch -d branchname # delete branch locally git push origin --delete

branchname # delete branch remotely git branch -a # list all remote

branches

Workflow:



Here index is staging area .Head is used to refer the last commit in the checkout branch.

Recovering lost commits:

- Reflog shows all the commits including git operations as well ..rebase , merge, branch created etc.... Reflog keeps track of every single change made in the reference of a repo.

Commands:

| | | |
|------------|---|---------------------------------------|
| git reflog | ▪ git reflog | # shows the history of HEAD |
| | ▪ show bugfix | # shows the history of bugfix pointer |
| | ▪ <u>How to recover the deleted branch ?</u> | |
| | git reflog | # gives all commit references |
| | git checkout -b deletedbranchname referncetothecommit ex: | |
| | git checkout -b feature HEAD@{4} | |

Bisect:

- *This is useful when someone committed error in the code and we are not sure which commit is causing the issue.*
- *Bisect command will checkout commits one by one [start, end] specified automatically until we find bad commit and mark it as bad. Interesting command.*

| | | |
|-----|--|--|
| | <u>Steps:</u> git bisect start | # this will initiate the bisect |
| | ▪ tool/command. | |
| git | ▪ bisect good hash | # we need to give the good commit id as start point |
| | ▪ git bisect bad [hash] | |
| | # we can give bad commitid if we are sure abt end pt, if we leave blank current commit will be end. # | |
| | Note: [] indicates the param is optional | |
| | ▪ Now, the start commit will be checkedout automatically by git and we need to test our scenario, if this commit is not causing issue , we need to mark it as follows. | |
| | ○ git bisect good | |
| | ▪ Now, the next commit will be checkedout automatically by git and again we need to test our scenario, if this is causing the issue, we need to mark it as follows. | |
| | ○ git bisect bad | |
| | ▪ similarly, after marking all commits as either good or bad .. git will provide us the details[commit, author, time etc....] of bad commit. | |
| | ▪ At this point we will be in detached HEAD state.. to get back to our current working branch we need to end the bisect process by following the command. | |
| | ○ git bisect reset | # terminates bisect session & checkout at wip branch |

Tagging:

- *This is useful when you want to save commit as checkpoint so that u can restore that checkpoint later when things go wrong. Ex: release v1 , release v2 etc In git we create a tag [checkpoint] for commits.*

Commands:

| | |
|---|---|
| ▪ git tag v1.0 | # tags the last commit as v1.0 |
| ▪ git tag v1.0 5e7a828 | # tags an earlier commit git |
| ▪ tag | # lists all the tags |
| ▪ git tag -d v1.0 | # deletes the given tag git |
| ▪ checkout -b branchname tagname | |
| #we can't checkout tags in it. so we need to create branch from tag. git push | |
| ▪ origin tagname | # to push tags to remote repo. git push |
| ▪ origin v1.0 | # pushes tag v1.0 to origin git push |
| ▪ origin —delete v1.0 | # delete the tag from remote |

Rewriting Git History:

Reference: <https://youtu.be/EIRzTuYln0M>

Amend:

| | |
|------------------------|--|
| git commit –amend | # this is helpful when u want staged files to be part of the last commit git |
| commit -amend –no-edit | # if we don't want to change the commit message |

Rewording:

- we can do in multiple ways either by using amend or rebase. Difference is that amend will work upon last commit whereas rebase we can work on any of the previous commits.
git commit --amend -m "correct message"
#Helpful when committed bad message & Modify last commit message without new commit git
rebase -i HEAD~2
#with rebase, we can act on previous commits as well .. HEAD~2 meaning want to act upon Head to 2 last commit

Deleting: if we want to drop or delete the commit itself. Git rebase -i HEAD~targetcommitnumber #Opens rebase interactive mode, prefix as drop.

Reordering: If we want to reorder the commits

Git rebase -i HEAD~2
#In interactive mode, just reorder the lines in VIM editor ..that's it DONE

Squashing:

- Combine multiple commits into one commit. If we have hundreds of commits history for a one feature branch, it will create mess to other developers in understanding the commits ..or it will pollute the commit history.. in this case we can squash all the commits into one commit.
- We can either use squash or fixup in interactive mode ..difference is that fixup will discard the commit messages, whereas squash preserve the messages.

Steps:

Git rebase -i HEAD~3

-i means interactive mode and HEAD~3 means we want to act upon last three commits from the HEAD

```
1 pick ee8ce13 Accessibility fix for frontpage bug
2 pick 4673264 Updated screenreader attributes
3 pick 5c4c543 Added comments & updated README
4
```

Replace 2 and 3 pick keyword with squash ..meaning combine 2 and 3 commit into 1st commit ..

before squash

after squash

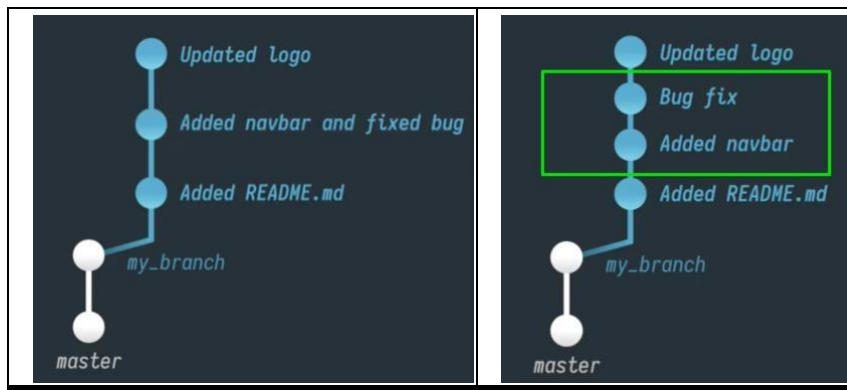
- How can we squash last N commits into one? One way is to follow above approach using interactive rebase, the other way is `git reset --soft HEAD~N && git commit -m "new commit message"` #N is no of commits to be combined

Splitting:

- if we want to split the commit into two different commits.

Steps:

- Git rebase -i HEAD~number
- In the interactive mode ..change the prefix of the target commit to **edit**, then `esc : w q`
- Note: rebase is in progress at this stage ..
- Next ..undo the staging files using the following command
 - Git reset HEAD^ # this will undo the staging files
- Continue git add and git commit ...as our wish to how many times we want to split that commit
- ... After committing, run following command. git rebase --continue ... DONE



Cherry-pick:

- *this is used when u accidentally committed the changes in wrong branch and want it to be in correct branch* Steps:
 - git `cherry-pick commitid` # apply this on the target branch
 - After cherry picking the commit, you need to remove i.e reset the commit from the wrong branch. 3 different ways to reset
 - git `reset --soft targetcommitid` [this will put targetcommitid files in staging area]git status for more details (or)
 - git `reset targetcommitid`
 - # this will put in working directory ..by default mixed reset is used] (or) ■
 - git `reset --hard targetcommitid`
 - # this will bring the exact state of targetcommitid including unstaged files. This may delete some wip changes.... To get them back use git reflog and git checkout targetcommitid
 - git `reset` # unstage all the changes ... optional git
 - git `clean -fd` # to clean the untracked files] ... optional git
 - git `reflog` # list all commits even deleted ... optional

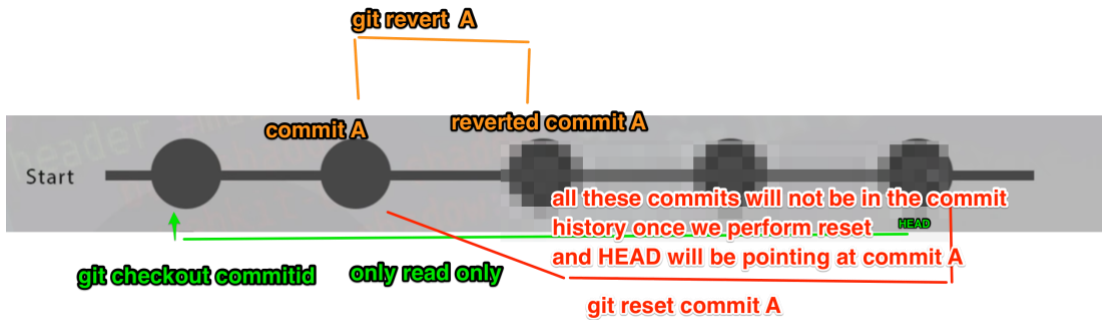
Stashing:

- *stashes can be applied in different branches .. for example ur working on master and stashed some changes ..now u want that changes in another branch ...you can apply the stash on new branch and it works* Steps:
 - git `stash save "message for recollect"` # to quickly switch to other branch and] git
 - git `stash push -m "New tax rules"` # same as above
 - git `stash list` # to see stash list] git
 - git `stash show stash@{1}` # shows the given stash git stash
 - git `show 1` # shortcut for stash@{1}
 - git `stash apply stashid` # to apply the stash, it will not delete the stash] git
 - git `stash pop` # applies the top stash and drop it from the stash list] git
 - git `stash drop stashid` # to drop/delete the stash] git stash clear
 - git `stash clear` # drops all stash]

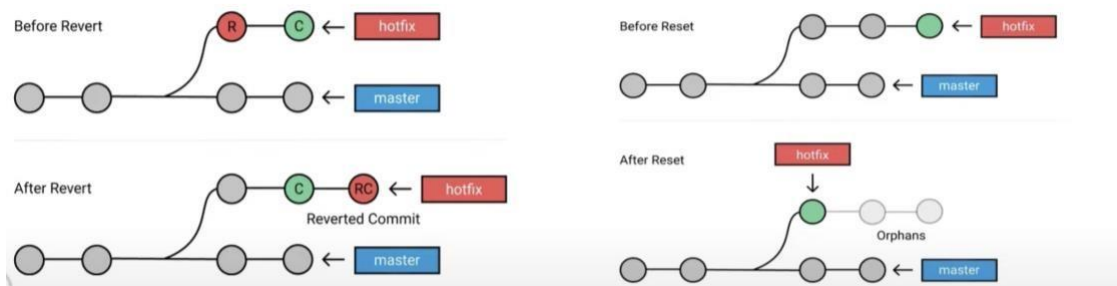
Undoing commits:

- By three ways we can undo things
 - git `checkout commitid`
 - #checkout commitid and work on it .. also detaches the head ■
 - git `revert commitid`
 - #used for reverting the commits, creates new commit and reverts the changes to previous, This is used to revert the commit which is not fixing the issues but others checkout the code on it.

- git reset commitid resets the commit, no history of commits is present

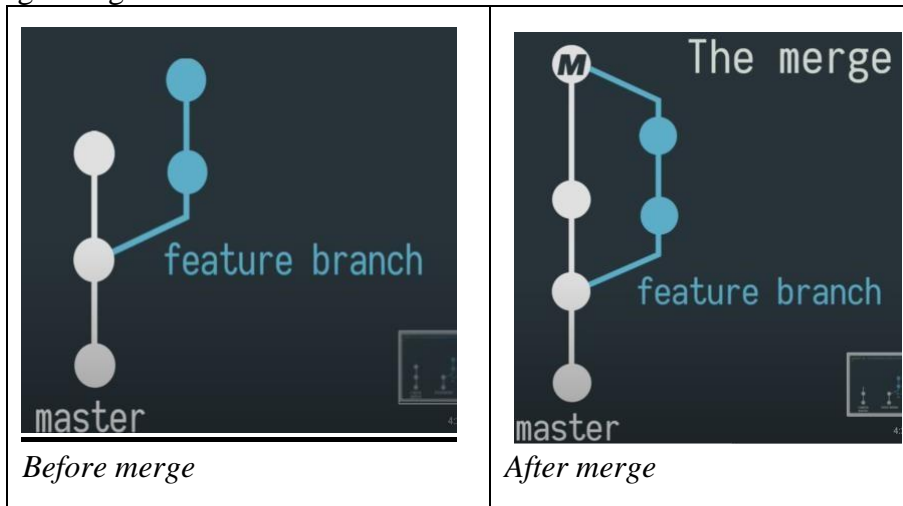


- Reset _ scraps and rewrites the commit history and it will take back to the last working state.
- Revert _ creates new commit and adds the history.



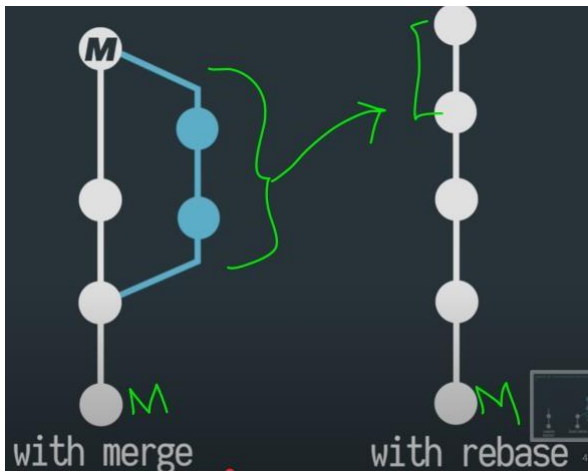
Rebase:

- The goal of rebase and merge is to merge the commits from feature branch to master branch.
- one of the difference between rebase and merge is ... rebase rewrites the commit history in order to produce the straight linear succession of commits. Whereas merge puts the fork history [will not rewrite the history] together again
- With merge:
 - git merge branchname



- With rebase: Steps:

- Git rebase master from featurebranch # this will point to latest
- Now checkout to master ..and git rebase featurebranch master
#this will pull the feature branch commits into master ...



Rebase another approach:

1. Checkout to main and create new branch with V1 as suffix and checkout to it.
 - `git checkout -b branchname-v1`
2. merge the old branch into new branch by using the following command
 - `git merge --squash branchname` and resolve the conflicts manually if any
 - if any problem occurs `git merge —abort`
3. Now commit changes
4. Now delete the old branch using following command
 - `git branch -D branchname`
 - Rename the `branchname-v1` to `branchname`
5. Open the bitbucket & decline the PRs if any and delete the remote branch [branchname] manually
6. git push and reopen the declined PR [it will sync all the changes and maintain commit history as well]
7. Done

Upstream:

- when u clone a remote repo and create local branch and work on it. And Now if u want to push the localbranch to remote repo.
- When you use `git push ...` you will get the following error ..
- ***fatal: The current branch beta has no upstream branch. To push the current branch and set the remote as upstream.***
- Meaning : your local branch is not present in remote repo .. so you need to upstream the localbranch or associate in remote repo. You can do in following ways: Use:
- `git push origin localbranch` # associate localbranch in remote repo and push the changes
- `git push --set-upstream origin localbranch` # set upstream in remote and push

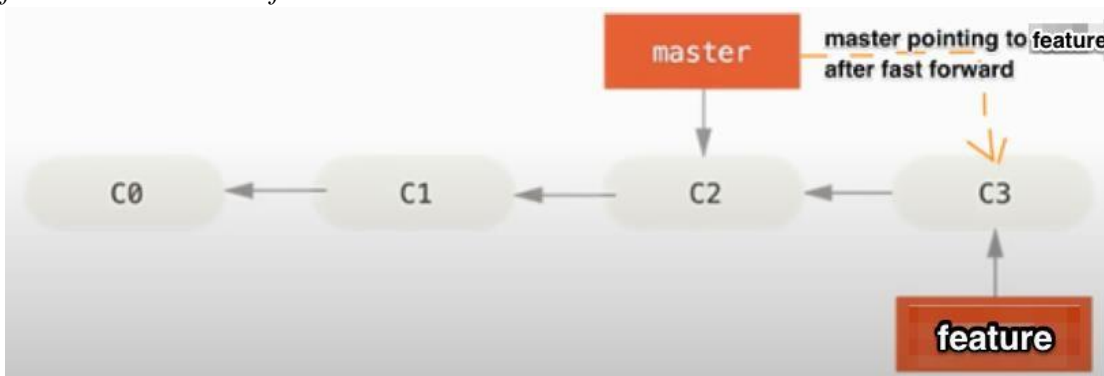
Bisect:

- This is useful when someone committed error in the code and we are not sure which commit is causing the issue. Bisect command will checkout commits one by one
- `[start, end]` specified automatically until we find bad commit and mark it as bad. Interesting command.
 - Steps: `git bisect start` # this will initiate the bisect tool/command.
 - `git bisect good hash` # we need to give the good commit id as start
 - `point git bisect bad [hash]`
 - # we can provide the bad commit id if we are sure as end point, if we leave blank current commit will be end. `[]` indicates the param is optional

- Now, the start commit will be checkedout automatically by git and we need to test our scenario, if this commit is not causing issue , we need to mark it as follows.
 - git bisect good
- Now, the next commit will be checkedout automatically by git and again we need to test our scenario, if this is causing the issue, we need to mark it as follows.
 - git bisect bad
- similarly, after marking all commits as either good or bad .. git will provide us the details[commit, author, time etc....] of bad commit.
- At this point we will be in detached HEAD state.. to get back to our current working branch we need to end the bisect process by following the command.
 - git bisect reset # terminates bisect session & checkout at wip branch

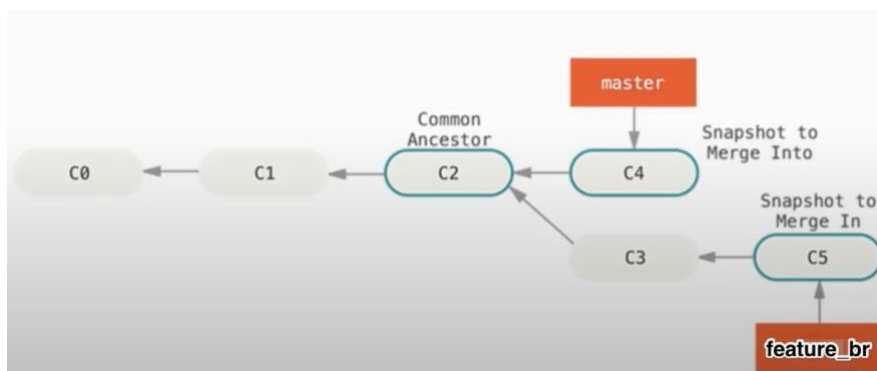
Fast-Forward Merge:

- We created feature branch from masterc2 and working on it, now when you merge feature, this will result in fast forward merge. That means as c2 is ancestor of c3, master will now be pointing to c3. Fast forwarded master to feature@c3



3-Way Merge or Recursive strategy:

- In this case as there is new commit in masterc4, and feature_br is not having the c4 latest master ... when we merge the feature_br into master, this merge follows recursive / 3 way strategy.



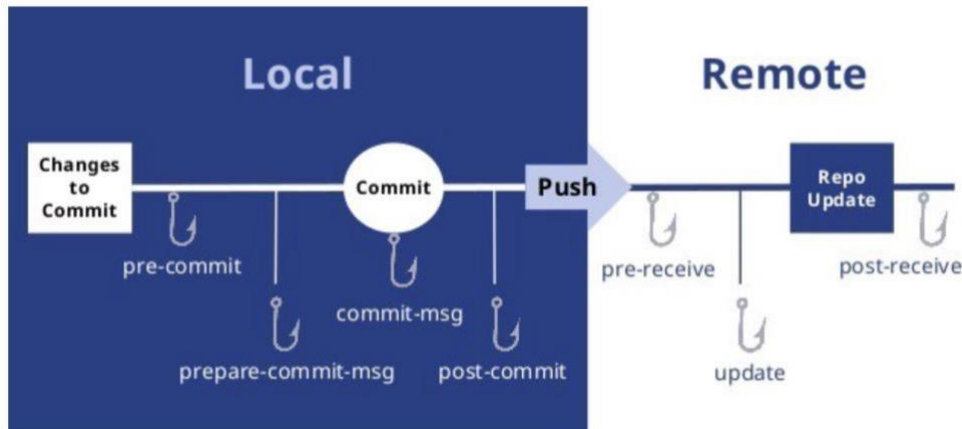
Git hooks:

- Git hooks allows to run the custom scripts whenever any of the git life cycle events occurs such as merge, commit, pushing etc..

```

~/my_repo$ ls -a
.  ..  .git
~/my_repo$ cd .git
~/my_repo/.git$ ls
branches  config  description  HEAD  hooks  info  objects  refs
~/my_repo/.git$ cd hooks
~/my_repo/.git/hooks$ ls
applypatch-msg.sample      pre-applypatch.sample      pre-rebase.sample
commit-msg.sample          pre-commit                  pre-receive.sample
fsmonitor-watchman.sample  prepare-commit-msg.sample  update.sample
post-update.sample         pre-push.sample
~/my_repo/.git/hooks$ vim pre-commit
~/my_repo/.git/hooks$ chmod +x pre-commit
~/my_repo/.git/hooks$

```



Git Internals(Internal working of git):

- For every git add . git creates one blob object which contains the content of the file.
- For every commit it creates , two objects tree and commit object.
- Tree object contains all related blob objects and commit object contains details of the commit. All these objects are stored in git as flat structure as below ... but the logic lies in how these objects are pointing to each other. That logic is stored in each commit object.

Commands:

- `git cat-file -p hash` # open the object
- `git cat-file -t hash` # gives type of object.

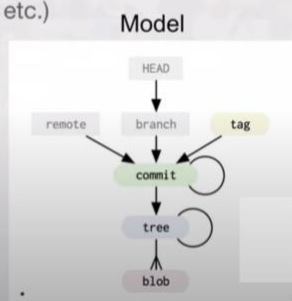
Object Types

- Commit - Author, message, pointer to a tree of changes
- Tree - Pointer(s) to file names, content, other trees
- Blob - Data (source code, pictures, videos, etc.)

These are stored as objects on the filesystem

Tags and Branches

- Pointers to commits (lightweight)
- Not full copies
- Allows a name for a commit
 - Contains additional meta information



```

Git-practice kants2$ git commit -m "made some changes"
[feature_updates/2942a1d] made some changes
1 file changed, 1 insertion(+)
create mode 100644 data.txt
Git-practice kants2$ git cat-file -t 2942a1d
commit
Git-practice kants2$ git cat-file -p 2942a1d
tree 570f5e59476bb0dc89d12c9c60efb98c748d5214
parent efa6761054f23f09154acd109d9c4e8ebcd1a31e
author Srikanth kante <Srikanth.Kante@in.pegacom> 1600594655 +0530
committer Srikanth kante <Srikanth.Kante@in.pegacom> 1600594655 +0530

made some changes
Git-practice kants2$ git cat-file -p 570f5e594
100644 blob 9cc2956aae936817090aa8c5778806078bac8e9b .gitignore
100644 blob 13212e765490d87cc075ed83a42763acbf4ebd9d data.txt
100644 blob c1b08eff03d8b6a13b1eda40eb14cd575950381 feature.htm
100644 blob d2a440721f26ed1dcc3cc4146fd759aee8edee36 index.js
100644 blob 2041af0941270d27448ccfca2784d1a57f0063d local_store
100644 blob ca19604f5783bdfadcf7432b875a69054ff7b69f master.js
100644 blob be9c03adcd7bbf830baebf9a671c3a2e1ede1f19 store.js
Git-practice kants2$ git cat-file -t 570f5e594
tree
Git-practice kants2$ git cat-file -p efa676105
tree 1f736b284a322881a6cbdd75fb555b37523fb84
parent af64625c1760580a6479af2a542746fd44623b25
author Srikanth kante <Srikanth.Kante@in.pegacom> 1600594401 +0530
committer Srikanth kante <Srikanth.Kante@in.pegacom> 1600594401 +0530

local store
Git-practice kants2$ git cat-file -p af64625c1760580a6479af2a542746fd44623b25
100644 blob 9cc2956aae936817090aa8c5778806078bac8e9b .gitignore
100644 blob c1b08eff03d8b6a13b1eda40eb14cd575950381 feature.htm
100644 blob d2a440721f26ed1dcc3cc4146fd759aee8edee36 index.js
100644 blob 2041af0941270d27448ccfca2784d1a57f0063d local_store.js
100644 blob ca19604f5783bdfadcf7432b875a69054ff7b69f master.js
100644 blob be9c03adcd7bbf830baebf9a671c3a2e1ede1f19 store.js
Git-practice kants2$

```

#1 : When you commit, git will respond with commit-id / hash. If you check the type, it is commit object. open the commit-id, it will give tree object and parent commit reference.

#2: open the tree object, it contains the blob object and file name. open the blob object and it contains the main content of the file.

#3: open the parent hash [previous commit is parent], it contains the previous state tree object and its parent.

```

DEBBUG CONSOLE  PROBLEMS  OUTPUT  TERMINAL
Git-internals kants2$ fswatch -n 1 -d find .
/projects/Git-internals/readme.txt 518
/projects/Git-internals/.git/objects/37/c33e42 a223dfde3f08ea2c7c3d3c2a725b506 514
/projects/Git-internals/.git/objects/37/1027
/projects/Git-internals/.git/objects/37/tmp_obj_7vld0 526
/projects/Git-internals/.git/index.lock 534
/projects/Git-internals/.git/index 528
/projects/Git-internals/.git/objects/9f/8b5a968ab1e3de3adb53461a672cd9548623 514
/projects/Git-internals/.git/objects/9f/1027
/projects/Git-internals/.git/objects/6c/1027
/projects/Git-internals/.git/objects/6c/tmp_obj_tdk84b 526
/projects/Git-internals/.git/COMMIT_EDITMSG 518
/projects/Git-internals/.git/objects/6c/1027
/projects/Git-internals/.git/logs/HEAD 518
/projects/Git-internals/.git/logs/refs/heads/master 518
/projects/Git-internals/.git/refs/heads/master.lock 534
/projects/Git-internals/.git/refs/heads/master 528
/projects/Git-internals/.git/HEAD.lock 522
/projects/Git-internals/.git/HEAD 592

Git-internals kants2$ ls
Git-internals kants2$ echo in git project >> readme.txt
Git-internals kants2$ git add .
Git-internals kants2$
Git-internals kants2$ git cat-file -t 37c33e42
blob
Git-internals kants2$ git cat-file -p 37c33e42
in git project
Git-internals kants2$ git commit -m "added readme"
[master (root-commit) 6ccd59d] added readme
1 file changed, 1 insertion(+)
create mode 100644 readme.txt
Git-internals kants2$
Git-internals kants2$ git cat-file -t 6ccd59d
commit
Git-internals kants2$ git cat-file -p 6ccd59d
tree 9f8b5a968ab1e3de3adb53461a672cd9548623
author Srikanth kante <Srikanth.Kante@in.pegacom> 1600598875 +0530
committer Srikanth kante <Srikanth.Kante@in.pegacom> 1600598875 +0530

added readme
Git-internals kants2$ git cat-file -t 9f8b5a96
blob
Git-internals kants2$ git cat-file -p 9f8b5a96
100644 blob 37c33e42a223dfde3f08ea2c7c3d3c2a725b506 readme.txt
Git-internals kants2$

```

fswatch is used to show the folder changes for every one sec

VIM Commands:

i # for insert mode ..to write esc : wq # write and quit dd
delete the line

:q # quit the VIM
vim filename # creates new file

for more VIM commands visit <https://www.fprintf.net/vimCheatSheet.html>

for Linux commands visit <https://maker.pro/linux/tutorial/basic-linux-commands-for-beginners>

