Masab Anees
Masab.anees019@gmail.com

Important Points for Git

Contents

Masab Anees
Masab.anees019@gmail.com

Git has 2 systems, a centralized and decentralized

Settings

We need to set the following

1. Name
2. Email
3. Default Editor
4. Line Ending

Setting apply to the following systems

1. System: applied to all the users.
2. Global Settings: applied to all repositories of current user.
3. Local Settings: applied to the current repository

Applying the Settings

- Set username
    a. git config –global user.name “masab anees”
- Set email
    a. git config –global email masaba019@gmail.com
- Set the default editor (I’m setting VS code)
    a. git config –global core.editor “core –wait”
- Setting the end of lines, this should be configurated when input the file in git
    a. Git config –global core.autocrlf true

We can always change the settings by the following command

a. git config –global -e

Get Help

- Get detailed help
    a. git config –help
- Get shorter summary
    a. git config -h

Masab Anees
Masab.anees019@gmail.com

Initializing a Repository

- The folder you want to make a repo, go into the folder and type
    a. git init
- To list the repo
    a. Ls -a
- Remove a directory
    a. Rm -rf .git

Git Workflow

- When you make a commit, it means you are taking the snapshot of the changes you make.
- Git has an intermediate stage which is staging area, that helps you to review your changes before the final commit.
- Git create the hidden directory in your folder that you can not access.
- If you don't want to commit the changes to, you unstage the files from staging area.

Real Example

- You have file in a folder name "file1". This will move the files to the staging area
    a. Git add file1 OR git add .
- Make a final commit, to COPY files from staging area to final directory as screenshot
    a. Git commit -m "my initial commit"

When you commit you staging area won't get empty, it's a stage area, where your next commit files or last committed files would be present.

Git doesn't track the file until the specially add it.

Staging File

- To see the status of the files when you add them in the git repository. If the color of the files are RED, it means they aren't in the staging area yet, because you haven't add it.
    a. git status
- Adding the files in the staging area
    a. git add . *# add all files present in the git repo*
    b. git add file1.txt file2.txt *# add to files with these names in the staging area*
    c. git add *.txt *# add files with the txt extension*

Masab Anees
Masab.anees019@gmail.com

- List the files present in the staging area
    a. git ls-files
- Stage and commit at the same time
- Git commit -a -m "YOUR COMMENT"

When you add the files in the staging area, the files go GREEN.

Committing Changes

When you have done with the changes in the file and want to commit/screenshot whatever work you have done. After running the command, the tree become empty.

a. git commit -m "this is my initial commit" # commit the files with the comment
b. git commit # this will open the default browser, so you can add lengthy comments

Committing best practice

- As you reach a stage you want to record, then you make a commit.
- If you fix the changes and make a commit while skipping the staging area
    a. git commit -am "Signup bug is fixed"
- If you want to remove the files from staging area, the changes are applied to both the working directory and staging area. It means there is no need to add this update into the staging area
    a. git rm file.txt
- Rename the file to "main.txt", the changes are applied to both the working directory and staging area. It means there is no need to add this update into the staging area
    a. git mv file1.txt main.txt

Ignore files

- There are many files that you want to ignore when committing such as log files. Follow the steps
    a. Echo logs/ > gitignore # add the directory to the .gitignore
    b. code .gitignore # open the file in the vs code and add files, directories and patterns that you want to ignore, add names like this:
        a. Logs/

b. Main.log
c. *.txt

c. git status # it will no longer says we have new directory called logs
d. git add .gitignore # add this file to the staging area
e. git commit -m “Add gitignore”

- If you already included the file that you want to ignore to track, then you cannot ignore that file. Follow the steps
    a. Git rm –cached -r bin/ # we need to remove the directory (bin/) from the staging area.
    b. Git commit -m “deleted the bin directory that was accidently committed”

Short Status

- You can view the short status, just need to learn the new sings here
    a. M -> file is modified, but the change is not in the staging area
    b. MM -> file in the working or staging area has been modified but doesn’t add the changes staging area yet
    c. M -> Modified file is added to the staging area
    d. ?? -> untracked file is created in the git directory or UNTRACKED FILE
    e. A -> created filed is added to the staging area to tracking

Viewing the staged and Unstaged Changed

- We need to review our code what is present in the staging area before committing because, we don’t want to add the broken code in the final commit
    a. Git diff –staged *# it will show the all changes that has been made in the file in staged that will go in the next commit.*
    b. Git diff *# lines in the red shows the changes has been made to the file but doesn’t add to the staging area, green shows the lines added to the staging area.*

Masab Anees
Masab.anees019@gmail.com

Want to made a diff visual in tools. First, we need to configure the vscode with the settings

a. Git config –global diff.tool vscode
b. Git config –global difftool.vscode.cmd "code –wait –diff $LOCAL $REMOTE"
c. Git config –global -e $ in the [difftool "vscode"] section add "$LOCAL $REMOTE" after –diff work in the vscode
d. Git difftool # to print the changes in the file that you want to track
e. Git difftool –staged # same comment in the above same command

View the History

The following command shows all the commits sorted from the latest to the earliest

a. Git log
b. Git log –oneline *# gives the short summary of the files committed*
c. Git log –oneline –reverse *# shows the latest commit on the top.*

View the content of the commit you added.

a. Git log –online
b. Git show <ID> or git show HEAD~2 # means move 2 steps forward from head

Want to view the files and directories stored in the commit

a. Git ls-tree HEAD~1
b. Git show <tree short ID> *# if you want to see the files changed or added in the specific tree iD*

View files in Unstaged Area

Lets say we have make a changes and added into the staging area and we want to UNDO it.

a. Git restore –staged file1.txt *# git took the copy of last committed file into the staging area*

Masab Anees
Masab.anees019@gmail.com

Discarding local changes

Make sure to understand this workflow. You modified the file and add it into the staging area, then you want to undo the changes has been made. Obviously, you want the previous version of your code, you type restore command in staging area to bring the copy of earlier committed into the staging area. If you want the earlier the last committed code, which is now in staging area into the local file you are working then type normal restore command

a. Git restore –staged file1.txt *# bring the last committed to the staging area*

b. Git restore file.txt *# bring the copy from staging to the local directory*

Delete the untracked file with ??

a. git clean -fd

Restoring the file of the earlier version

Let's say you have deleted the file and commit the changes. You want to bring that file back. Let's delete it first and bring it back from git database

a. Git rm -f file1.txt *# delete files from the staging area, hence completely removed the file*
b. Fit restore –source=HEAD~ file1.txt *# bring back the file to the staging area*

Creating the alias

You can create the alias of the commands that you normally run

a. Alias log="git log –oneline"

Masab Anees
Masab.anees019@gmail.com

Branches

Branches allow us to work with the same file in difference branches and pointers. Whenever you create a branch you create a new pointer to the HEAD, which is pointing to the master branch so you can work in parallel independently. Branches is the separation of version of the same file. We can have multiple branches like we can have production branch, development and bug fixes branches.

Create the branch, but HEAD initially point to the master

a. Git branch <branch_name>

To view the branches

a. Git branch

View the branches in graph

a. Git log –all –decorate –oneline –graph

Change the branch and head start pointing your branch

a. Git checkout <branch_name>

If you want to see the branches happen to the file when another is created. Checking the diff between the master and SDN branch

a. Git diff master..SDN

Fast forward merge, where we are bringing master to the branch, in this case to SDN.

a. git checkout master *# bring pointer to the master*
b. git merge SDN *# write the branch name to which to merge*

To check which branches has been merged

a. Git branch –merged

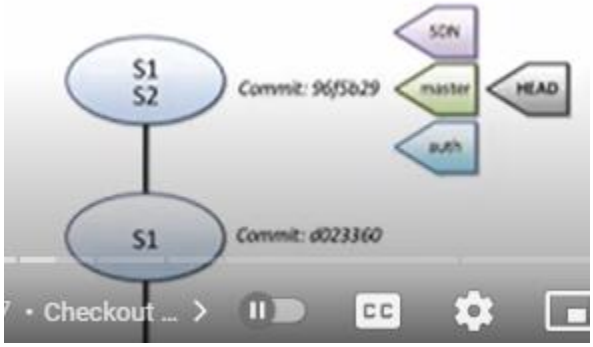
Delete the branch SDN, since it has been merged with master, so we can safely delete SDN now.

a. Git branch -d SDN

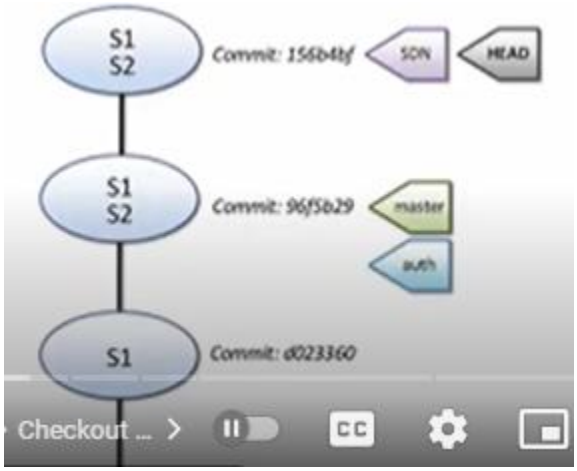
Masab Anees
Masab.anees019@gmail.com

Understanding with the diagram of different branch concept

Lets say, I'm on master branch and make a commit of 2 files present in the staging area. 2 person create 2 branches of name "SDN" and "auth". Now the SDN and auth pointer point to the same committed node and head pointer is pointing to the master branch



SDN person change the head pointer to SDN branch and make a commit. A new node is created for that commit on SDN branch and head is pointing to the SDN, while master and auth is still pointing to the previous node.



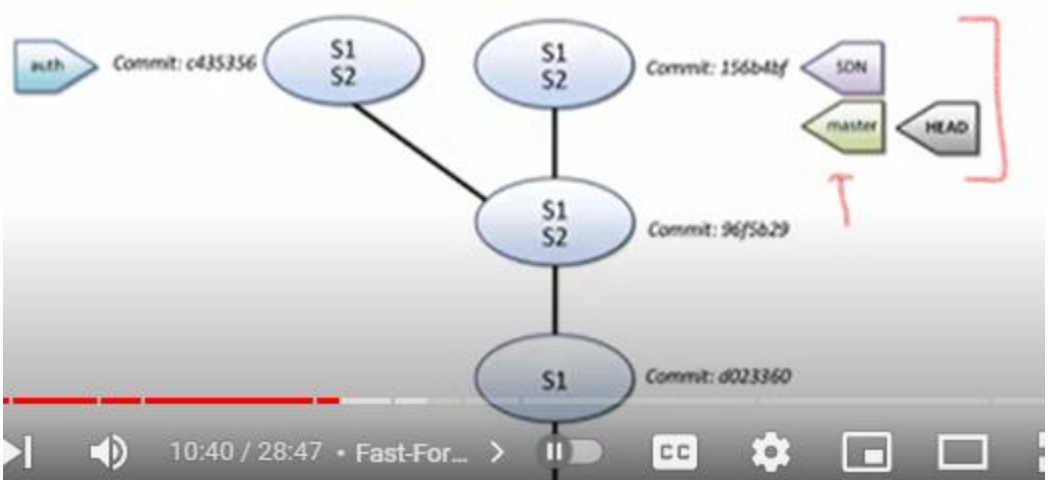
Masab Anees
Masab.anees019@gmail.com

Auth person changed the pointer and start working on the last committed version and make some changes and commit again. Now head is pointing to the auth

Note: The versions that SDN create or make won't be visible to the auth and master until he merged.

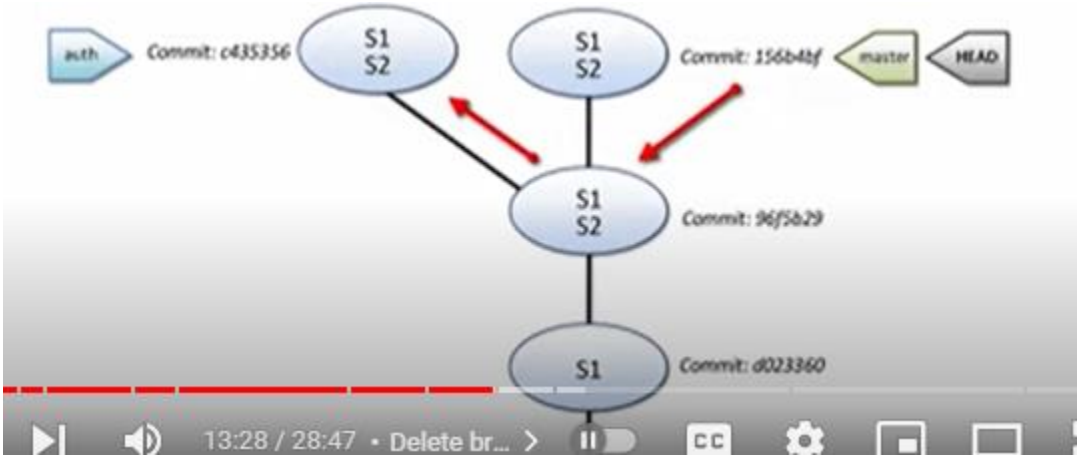
Fast Forward Merge

In this example SDN ahead of master branch, with fast forward we can bring master to the SDN branch, while head is pointing to the master. For this to happen we need a direct path

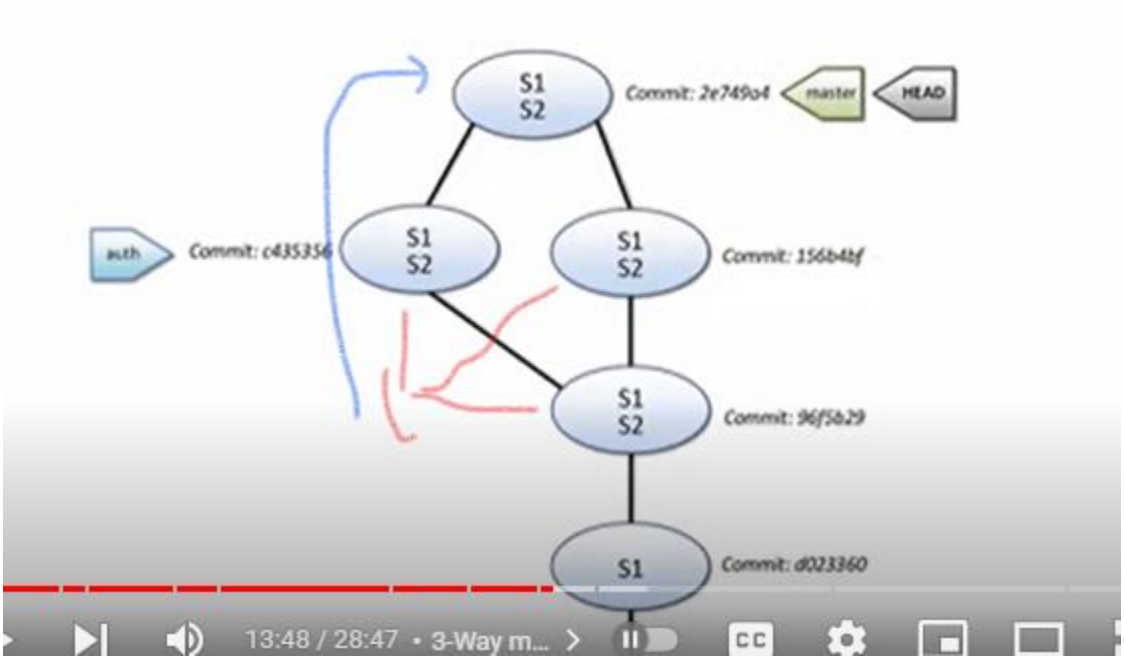


Masab Anees
Masab.anees019@gmail.com

- Once we have done with the branch and merged with the master, we delete that branch. First check with branched as been merged, in this case master and SDN is merged, we delete SDN here. We cannot delete the branch with has not been merged, in this case “auth”.
- There are scenarios where we worked on some branch and we no longer needed that, we simply delete the branch without merged with “-D” option.
- 3 way merge. If you want to merge the “auth” branch this time, we need a 3 way merge



These 3 nodes will merge into 1 (pointing with blue)



Masab Anees
Masab.anees019@gmail.com

Conflicts

This will happen when you make a different change in the same line in file. Now when you merge, it become the conflicts. For this case Git can't guess for us which commit to keep. Let's say if there is an IP written in the file and both branches change to the same IP, then there won't be any conflicts.

We need to follow the steps if we got the conflicts in the merging.

1. Git merge –abort *# we need to abort the merge first*
2. Open the file and resolve the error. Both branch name has been written in the file which shoes where the commit occurs.
3. Git add <file_name> *# added file to the staging area, but still file is under commit*
4. Git commit *# file successfully merged without conflicts*

Detached Head

Sometimes what happen, when we checkout any commit in the log, heads become detached from master and not pointing to any branch. Way to resolve this is we create the branch there and starting pointing head to that branch, again we move back to the master branch.

1. Git log
2. Git checkout <log first 6 digits ID> *# this will be detached*
3. Git branch <branch_name>
4. Git checkout master *# change the head pointer to the master branch*

Git Stash

We always have clean working tree and branches and not any mortified file in the staging area. Sometime we don't have a clean tree and we block from changes branches and merge. This generates the clean stage for you. Stashes are present in the

1. Lets suppose we make a change in the file and doesn't add the file in the staging area and we are on any branch other than master for this example.
    a. Git checkout master *# this will give me the error, I left unstagged modified file*
    b. Git stash *# this will keep your changes in DRAFT so you can work on it later*
2. Git stash list # display all the stashed files

3. Git stash list -p # we display the changes have been made to the file
4. To apply the most recent stash
    a. Git stash apply
    b. Git commit -a -m "Git Stash has been applied" *# commit the changes*
5. Apply with label names
    a. git stash list
    b. git stash apply <label> *# get label from above command*

6. Adding comment to the stash
    a. Git stash save "add yellow label"
    b. Git stash list