GIT and GITHUB

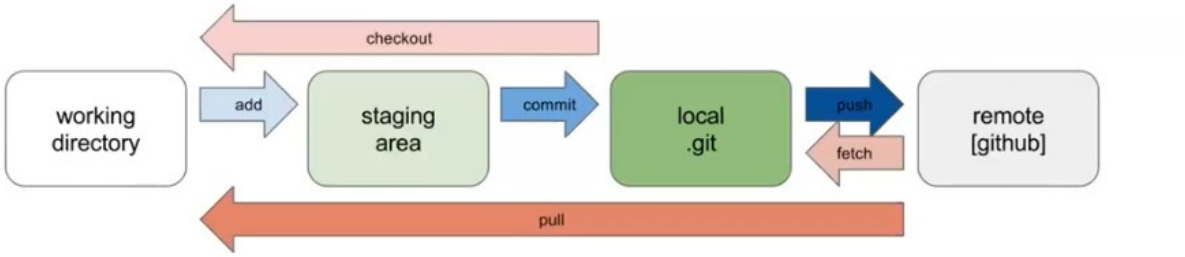
1. basics commands
2. git log
3. git diff command
4. git rm command
5. undo changes using git checkout command
6. git reference (master and head)
7. git reset
8. git ignore
9. git branching and commands
10. merging and conflict
11. deleting branches
12. rebashing
13. git slash
14. git remote commands
15. git taging

What is git? git vs github?

What is GIT?

- **Git is Distributed Version Control System Tool.**
- **Git is not acronym and hence no expansion.But most of the people abbreviated as "Global Information Tracker".**
- **GIT is developed by Linus Torvalds, who also developed Linux Kenel**

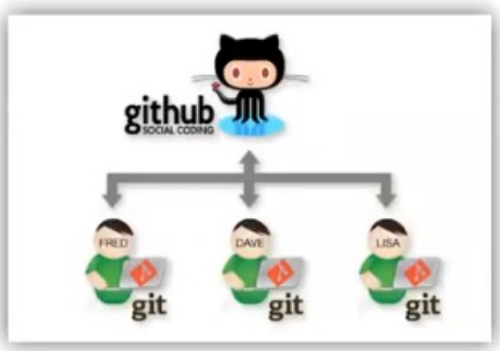
- **Git** is a revision control system used to track changes in computer files. It's a tool to manage your code & file history while coordinating work remotely on those files with others.

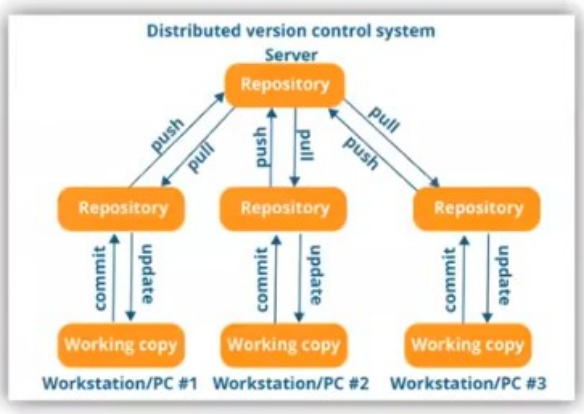


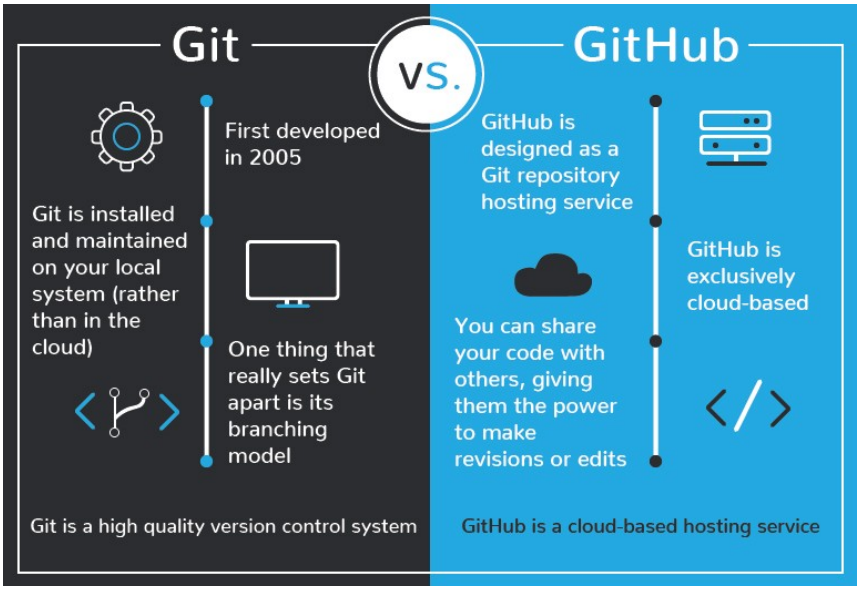
GIT is **distributed version control system**, github is an **cloud based hosting service**

https://blog.devmountain.com/git-vs-github-whats-the-difference/

GitHub is a hosting service for git repositories. Git is the tool, while GitHub is the service to use git.



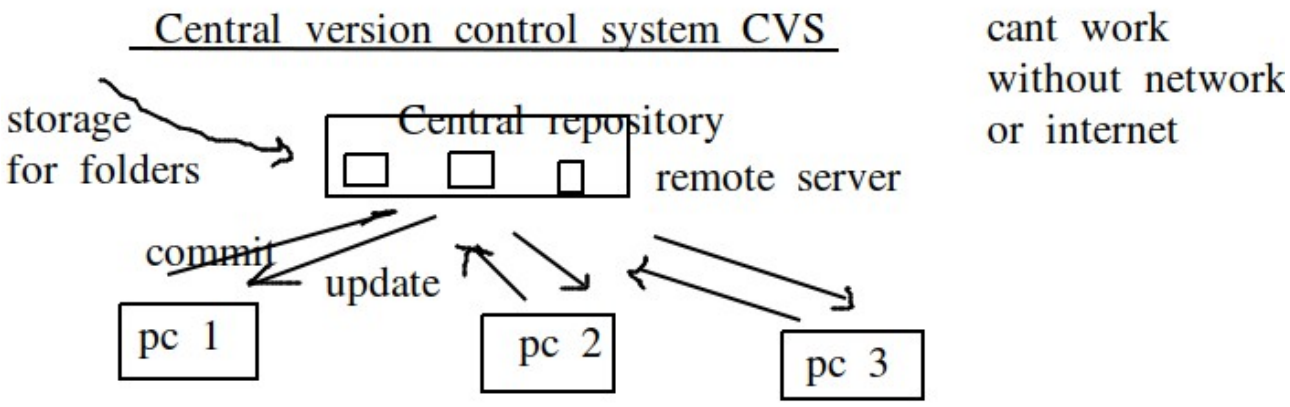




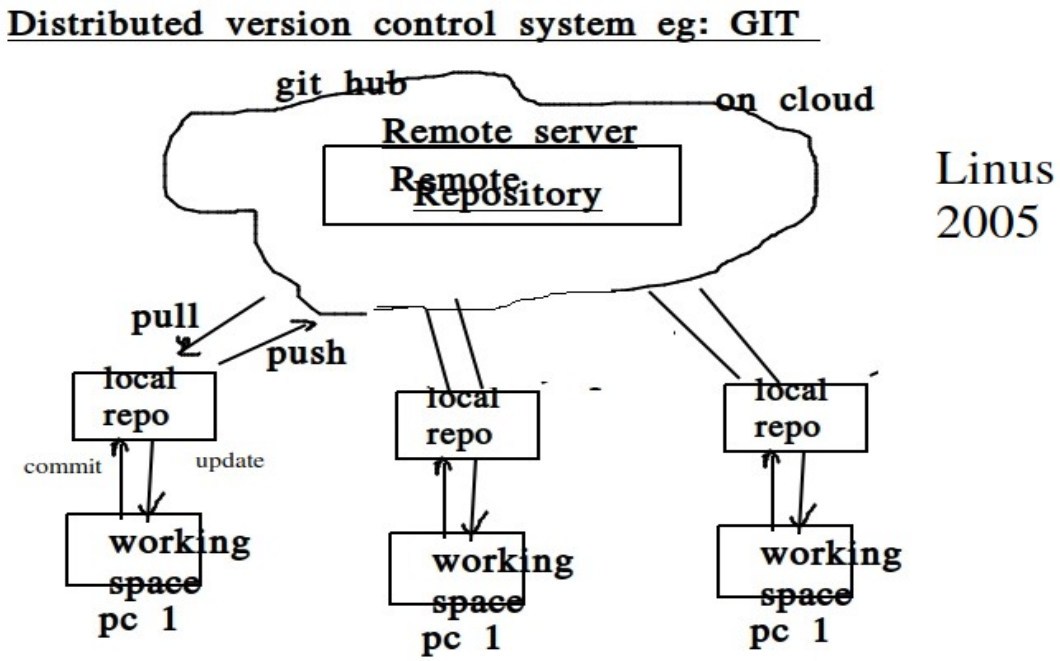
Type of version control system:

1. centraiized version control system CVS eg SVN
2. Distribteud version control system eg: GIT

How CVS works?



Problem with CVS?
1. Not locally available, we need to connect to network or internet always. Slow performance
2. Single point of failure



In git every contributer has a local copy or “clone” of the main repository
ie everybody maintain a local repository of there own which contain all the files and metadata present in teh the main repository

CVS vs Distribteud version control system GIT

CVCS

- In cvcs a client need to get local copy of souce from server, do the changes and commit these changes to central server

- CVCS easy to learn and setup

- working on branches is difficult in CVCS and leads to many confilicts

- CVCS dont provice offline access

DVCS

- In DVCS each client can have a local repo as well and have a complete history on it client need to push the changes to branches which will then be pushed to server repository

- DVCS system are difficult for beginer, multiple commands need to remember

- working on branches is easier in DVCS, less conflict

- DVCS works fine for offline access as every dev have copy of whole project on his local system , interet is only required for pull/push operation

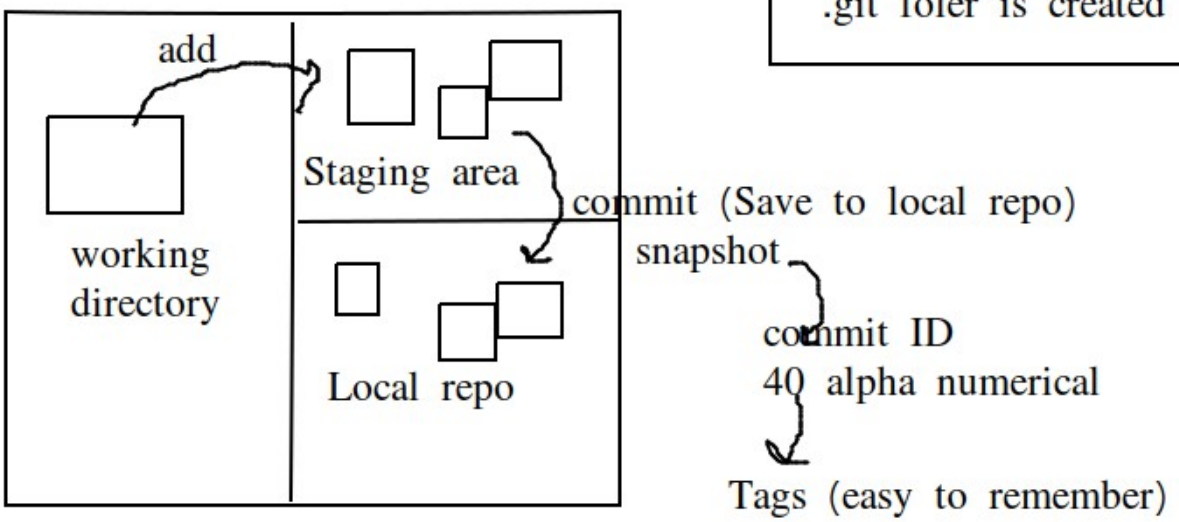
- DVCS faster as dev use his local copy most of the time, dev has to intract with central repo not too frequently

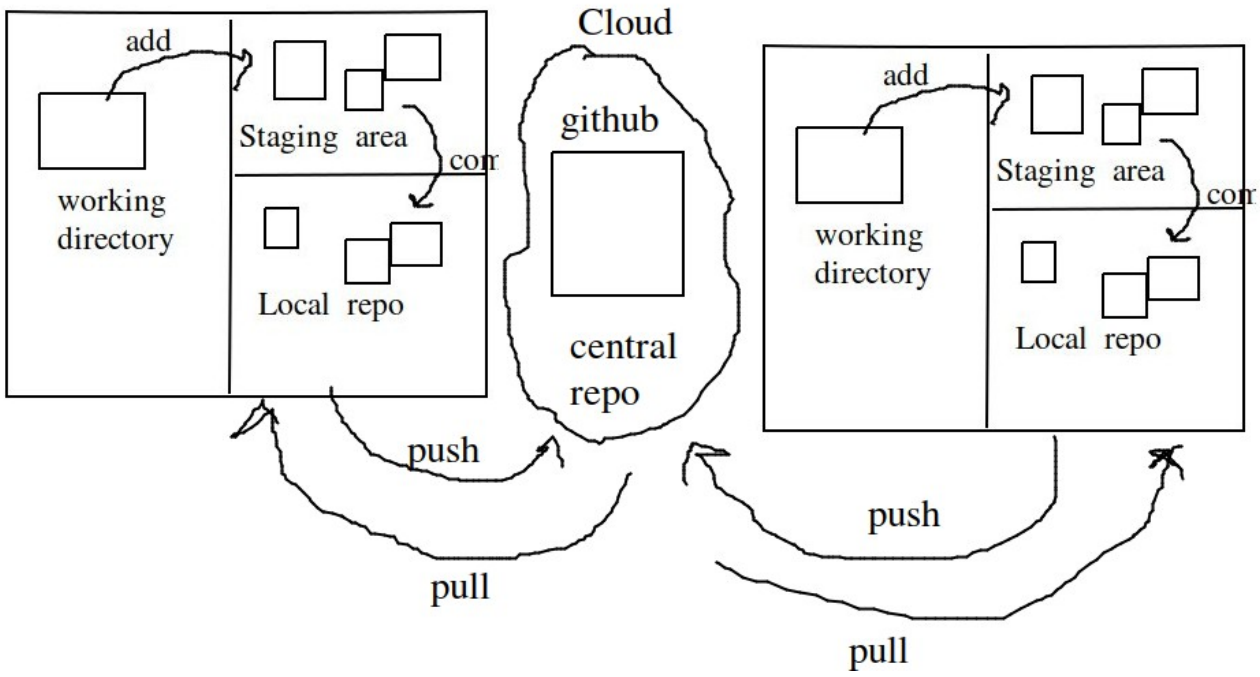
- If DVCS server is down still dev can work on his copy

Stages of git and terminology

1. working space / working directory
2. staging area
3. local repository

git init => initilize an repo
.git foler is created





When dev2 pull code from github copy of code would be there in working directory, staging area and local repository

Dev2 add his own code and finally commit code to local repository and push to central repository, Dev 1 next day pull and get updated code

All changes are maintained with all version history is maintained in git

GIT important terminology

Repositry

- Repository is a place where you have all your code
- Repository is a kind of folder related to one product
- Changes are personal to that particular repository

server:

- it store all Repository and contain meta data also

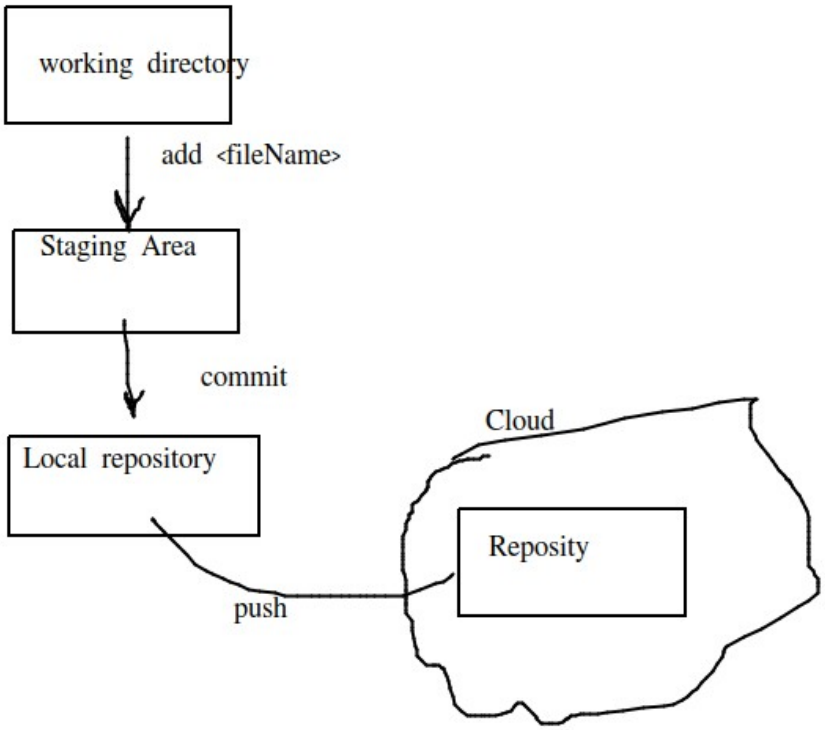
Working directory/ workspace

- Where you see files physicall and do modification
- At a time you can work on a particular branch in workspace only

How git is different from Older CVS

In older version system CVCS,there is no concetp of staging area, developer marks modification and commit there changes directly to the repository, so because CVCS dont use concept of staging area, cvcs need to track all files that is used by that developer :(

GIT use different apporach, git only track files that are added to staging area, whenever we do commit an operation git looks for the files present in the staging area and that files only considered for commit and not other modified files present in working directory



Commit id/ version-id/ version

- Reference to identity each changes
- TO identity who changes which files

Tags

- Tags assigning some meaningful name with a specific version in the repositry, once a tag is created for a particualar save, even if you create a new commit, it will not be uudated
- Used for versioning V-x.y.z
-

Snapshots

- represent some data fo particular time
- it is alwasy incremental ie it store the changes (append data) only, not entrire new copy

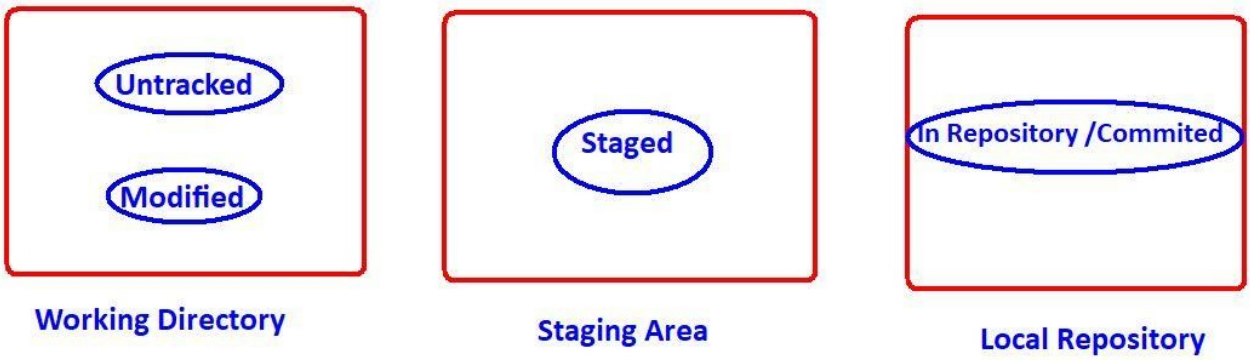
Commit

- commit will chnages in repository, you will get and unique commit id (40 alpha numeric code)
- It uses sha-1 checksum concept, provide code integrity/security, sender and reciver checksum must be some
- even if you changes one dot, commit id will be changed
- It actually help you to track changes
- commit is also named as SHA1 hash

Features and Architecture of GIT

Staging Area/index area.

There is logical layer/virtual layer in git between working directory and local repository.



Working Directory ==> Staging Area ===> Local Repository

We cannot commit the files of working directory directly. First we have to add to the staging area and then we have to commit.

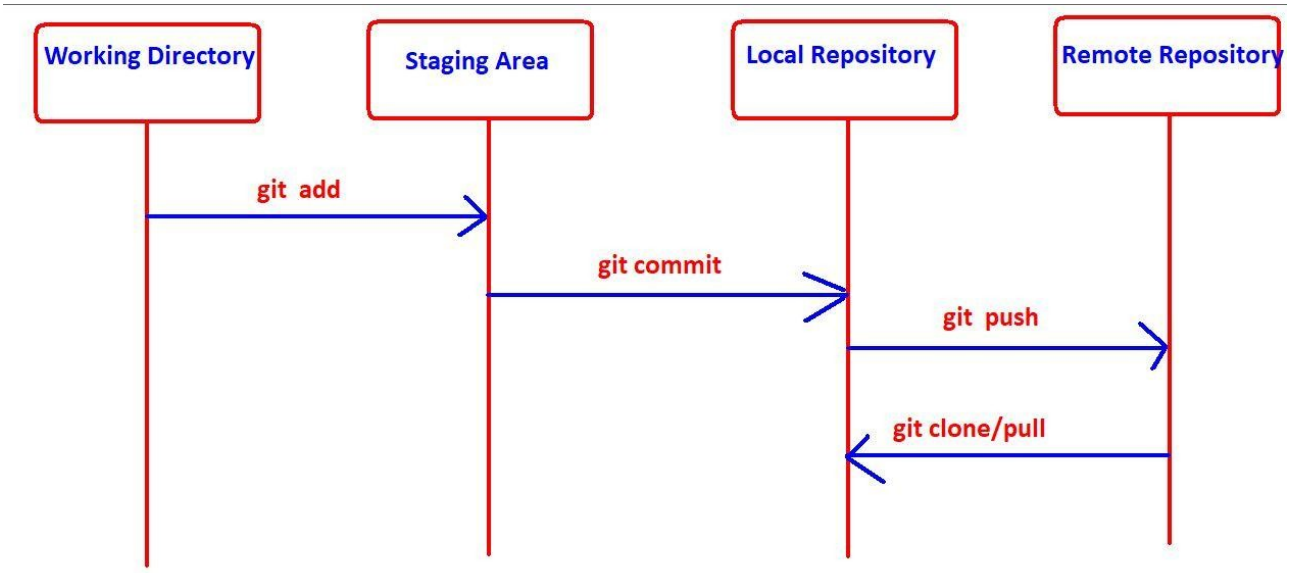
This staging area is helpful to double check/cross-check our changes before commit.

Git stores files in repository in some hash form, which saves space
GIT will uses internally snapshot mechanism for this. All these conversions and taking snapshots of our data will be happened in staging area before commit.

Branching and Merging:
We can create and work on multiple branches simultaneously and all these are branches are isolated from each other.
It enables multiple work flows.

We can merge multiple braches into a single brach. We can commit branch wise also.

GIT Architecture:



GIt contains 2 types of repositories:
1) Local Repository
2) Remote Repository

For every developer, a separate local repository is available. Developer can perform all checkout and commit operations wrt local repository only.

To perform commit operation, first he has to add files to staging area by using git add command, and then he has to commit those changes to the local repository by using git commit command.

git add => To add files from working directory to staging area.

git commit => To commit changes from staging area to local repository.

git push => To move files from local repository to remote repository.

git clone => To create a new local repository from the remote repository.

git pull => To get updated files from remote repository to local repository.

Life Cycle of File in GIT

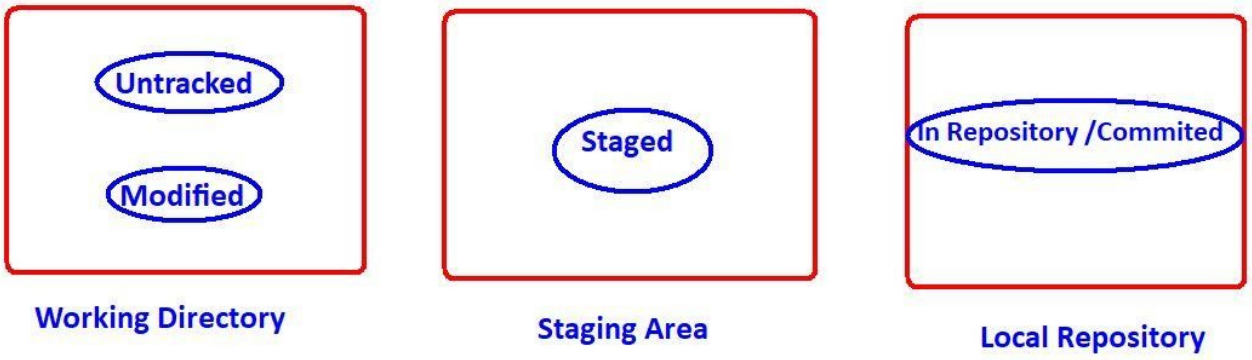
Every file in GIT is in one of the following states:

1) Untracked:
The files which are newly created in working directory and git does not aware of these files are said to be in untracked state.

2) Staged:
* The files which are added to staging area are said to be in staged state.
* These files are ready for commit.

3) In Repository/ Committed:
Any file which is committed is said to be In Repository/Committed State.

4) Modified:
Any file which is already tracked by git, but it is modified in working directory is said to be in Modified State.



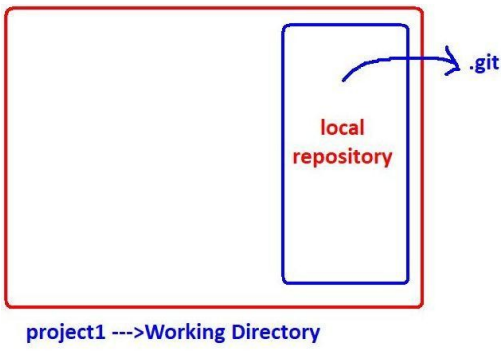
1. GIT basics commands

git status

fatal: not a git repository (or any of the parent directories): .git

git init

Initialized empty Git repository in /home/raj/Desktop/gitdemo/project1/.git/



git status

On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to track)

Now create some files
cat >file1
this is file 1
cat >file2
this is file 2

git status

On branch master
No commits yet

Untracked files:
file1, file2

git add file1, file2

git status

On branch master
No commits yet

Changes to be committed:
new file: file1
new file: file2

git commit -m "first commit"

*** Please tell me who you are.

Run

git config --global user.email "you@example.com"
git config --global user.name "Your Name"

to set your account's default identity.

Configure username and password

git config --global user.email "rgupta.mtech@gmail.com"
git config --global user.name "rgupta00"

git commit -m "first commit"

[master (root-commit) bf0acc9] first commit
2 files changed, 2 insertions(+)
create mode 100644 file1
create mode 100644 file2

git status

On branch master
nothing to commit, working tree clean

Now update file1:

cat >> file1
this is some update to file1

git status

On branch master
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)
modified: file1
no changes added to commit (use "git add" and/or "git commit -a")

git add .

git commit -m "some changes to file1"

2. git log : get log information:

git log command is used to see history of all commits in local repository most commonly command

git log . => **getting all log information**
git log file1 => **only commit related to file1**
git log --oneline => **oneline message, useful to identity commit based on messages**

git log

commit 0a82246afbfba89105ffa95a0b2cc65ccfc0010c (HEAD -> master)
Author: rgupta00 <rgupta.mtech@gmail.com>
Date: Thu Aug 6 11:59:07 2020 +0530

some changes to file1

commit bf0acc933722c9740521ed2dc98c9c899365dc99
Author: rgupta00 <rgupta.mtech@gmail.com>
Date: Thu Aug 6 11:56:59 2020 +0530

first commit

git ls-files:

shows files in staging area

file1
file2

git log file1

commit 0a82246afbfba89105ffa95a0b2cc65ccfc0010c (HEAD -> master)
Author: rgupta00 <rgupta.mtech@gmail.com>
Date: Thu Aug 6 11:59:07 2020 +0530

some changes to file1

commit bf0acc933722c9740521ed2dc98c9c899365dc99
Author: rgupta00 <rgupta.mtech@gmail.com>
Date: Thu Aug 6 11:56:59 2020 +0530

first commit

git log --oneline:

handy command to see oneline log messaege show first seven char of commit id

0a82246 (HEAD -> master) some changes to file1
bf0acc9 first commit

git log -n 2
get log for latest first two commit

commit 0a82246afbfba89105ffa95a0b2cc65ccfc0010c (HEAD -> master)
Author: rgupta00 <rgupta.mtech@gmail.com>
Date: Thu Aug 6 11:59:07 2020 +0530

some changes to file1

commit bf0acc933722c9740521ed2dc98c9c899365dc99
Author: rgupta00 <rgupta.mtech@gmail.com>
Date: Thu Aug 6 11:56:59 2020 +0530
first commit

git log --grep "firstr" --oneline

git log --since="5 min ago"

git log --author =rgupta00 –oneline

git log --max-count=3

3. git diff command

lets say afte commiting we change file in working directory then content of staging area will be different from local repository

we want to see the difference bw working directory and staging area
=> git diff command

git diff file1

Will give diff bw working dir and staging area

```
diff --git a/file1 b/file1
index 6564281..171840f 100644
--- a/file1
+++ b/file1
@@ -2,3 +2,4 @@ this is file 1
 this is some update to file1
 this is added to file1
 another line is added to file1
+this is another line
```

diff --git a/file1 b/file1:

a/file1: source copy ie staging area
b/file1: destination copy ie working copy

staging area ----> working copy

index 6564281..171840f 100644

6564281: hashcode of soruce file
171840f: hashcode of destination file
100644: GIT file mode
100: first 3 digits, represent type of file,ASCII text file
644: next 3 digits rep file permission
644-> rw-r--r--

--- a/file1

---- a/file1 is missing some lines (staging area)

+++ b/file1

+++ b/file1 is having some extra data (working dir)

@@ -2,3 +2,4 @@ this is file 1

this is some update to file1
this is added to file1
another line is added to file1

+this is another line

=> if any line prefixed with space it is unchanged
if any line is prefixed with + then it is added in destination copy

if any line is prefixed with - then it is removed in destination copy

git rm command:

git rm file1.txt => it will remove file from both working and staging area

git rm --cached file1.txt => it will remove file only from and staging area and not from working dir

rm file1.txt => it will remove file from only from working area not from staging area

As files are alread in local repo can revert back! We can only restore commited files

commit operation is requried even in case of removing files, if files is alread tracked by git.

undo changes using git checkout command

git checkout command? something like undo operation

purpose: we can use checkout command to ignore/discard unstaged changes (not added to staging area) in the tracked files of working directory.

git checkout – file1.txt

to discard unstaged changes in working directory of tracked file

Git References or refs

(master and HEAD)

For most of the commands (like git log, git diff etc) we have to provide commit id as argument. But remembering commit id is very difficult to remember, even 7 characters also.

Git provides some simple names for these commit ids. We can use these names directly.
These are just pointers to commit ids. These sample names are called references or refs.
References are stored in **.git/refs** directory as text files.

There are multiple types of references like **heads,tags and remotes.**

The most recent commit id => master or HEAD

What is master?
$ git status
On branch master

1) master is the name of the branch.

2) It is a reference(pointer) to last commit id. Hence where ever we required to use last commit id, simply we can use reference master.

3) This information is available in .git/refs/heads/master file.

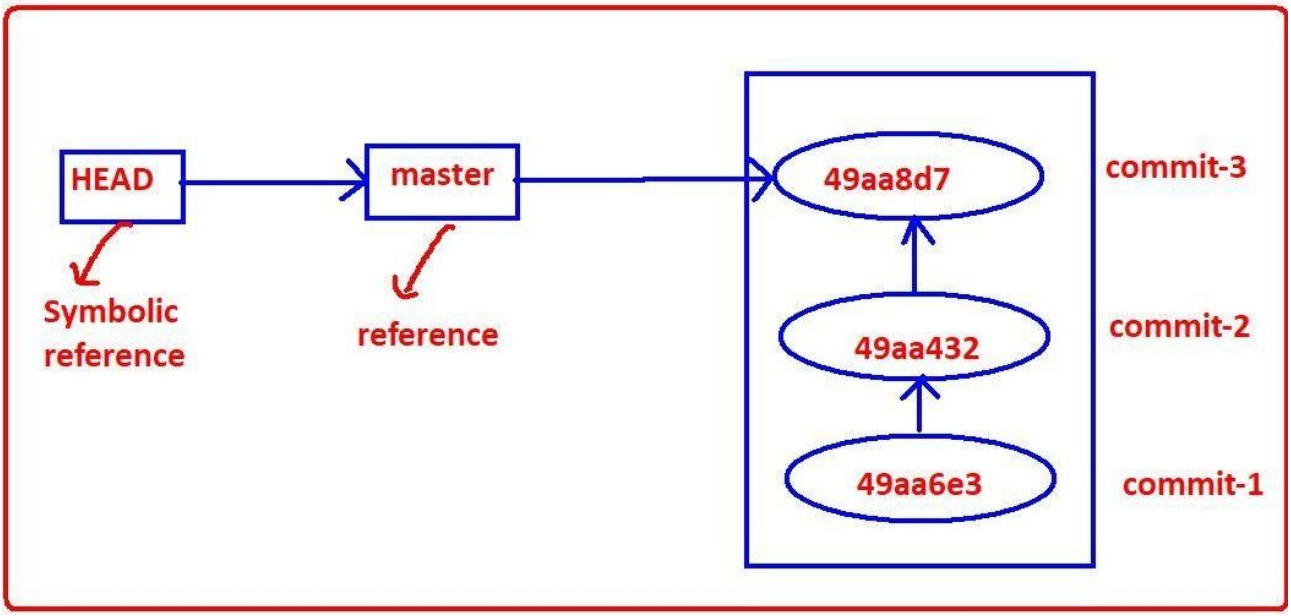
The following two commands will produce same output.
$ git log 49aa8d7
$ git log master

What is git Head?

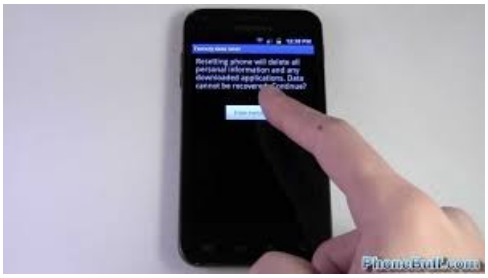
HEAD is a reference to master.

If any reference pointing to another reference, such type of reference is called symbolic reference. Hence HEAD is symbolic reference.

By default HEAD is always pointing to branch(master).
$ git log --oneline
49aa8d7 (HEAD -> master) both files added
HEAD is stored in root of .git directory but not in .git/refs directory.



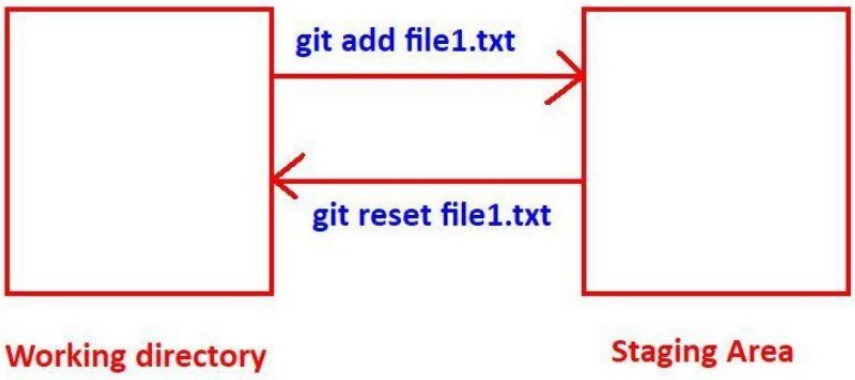
Git reset Command



git reset command is just like reset settings in our mobile.

There are 2 utilities of git reset command.
Utility-1: To remove changes from staging area
Utility-2: To undo commits at repository level

To remove changes from staging area (It is opposite to git add command.)
Changes already added to staging area, but if we don't want to commit, then to remove such type of changes from staging area, we should go for git reset.
It will bring the changes from staging area back to working directory.



git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified: file1

raj@raj-Aspire-E5-575G:~/Desktop/devToolsStuff/gitdemo$]git
Now run the command:
git reset file1
Unstaged changes after reset:

```
M       file1
raj@raj-Aspire-E5-575G:~/Desktop/devToolsStuff/gitdemo$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   file1

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

Utility-2: To undo commits at repository level

git reset <mode> <commitid>
Moves the HEAD to the specified commit, and all remaining recent commits will be removed.
mode will decide whether these changes are going to remove from staging area and working directory or not.

Example:git reset --mixed 86d0ca3 (mixed is default mode, optional)

Ignoring unwanted Files And Directories by using .gitignore File

It is very common requirement that we are not required to store everything in the repository.
We have to store only source code files like .java files etc.
README.txt Not required to store
log files Not required to store

We can request git, not to consider a particular file or directory.
We have to provide these files and directories information inside a special file .gitignore (without extension)

.gitignore File:
We have to create this file in working directory.
Don't track abc.txt file

abc.txt
Don't track all .txt files
*.txt
Don't track logs directory
logs/
#Don't track any hidden file
.*

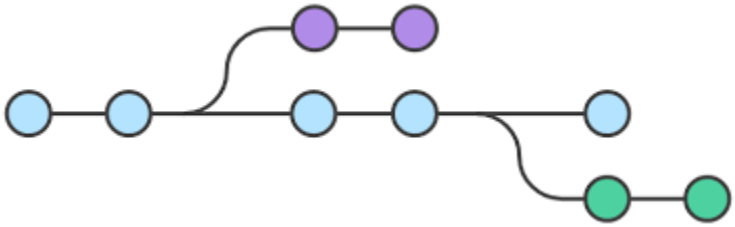
Example:
$ touch a.txt b.txt Customer.java
$ mkdir logs
$ touch logs/server.log logs/access.log

$ git status
On branch master
Untracked files:
(use "git add <file>..." to include in what will be committed)
Customer.java
a.txt
b.txt
logs/

Now lets create . **Gitignore file**

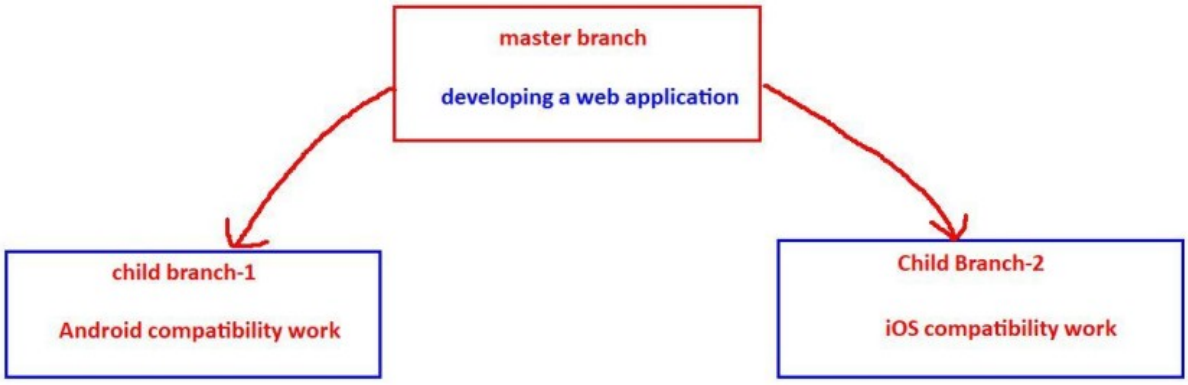
Don't track a.txt
a.txt
#Don't track all .txt files
*.txt
#Don't track log files
logs/
#Don't track any hidden file
.*

Branching And Merging



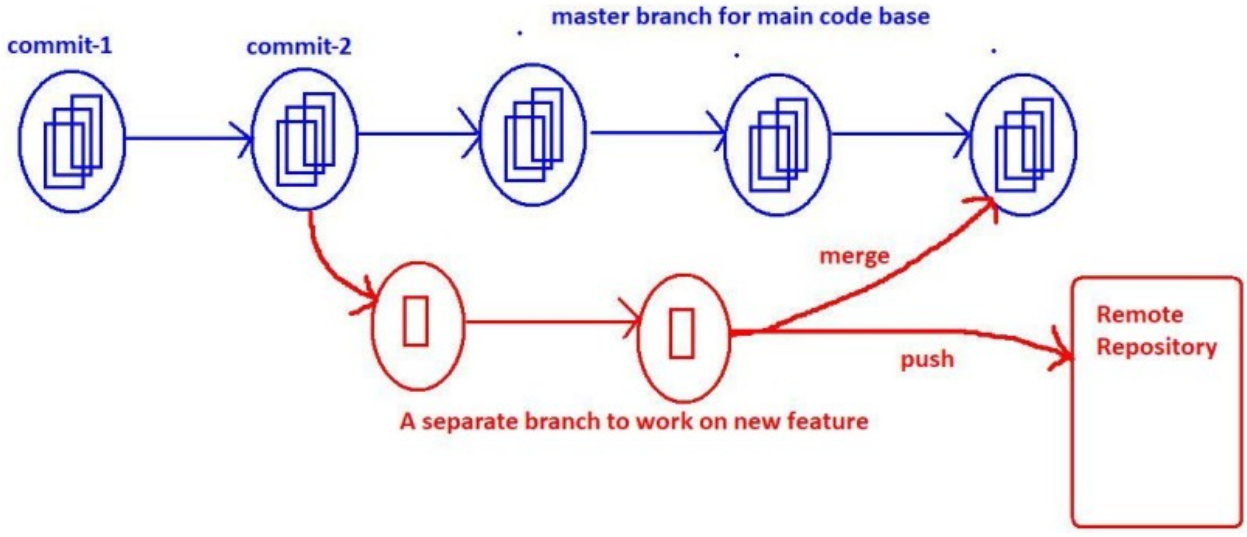
Branching is one of very important concept in version control systems.

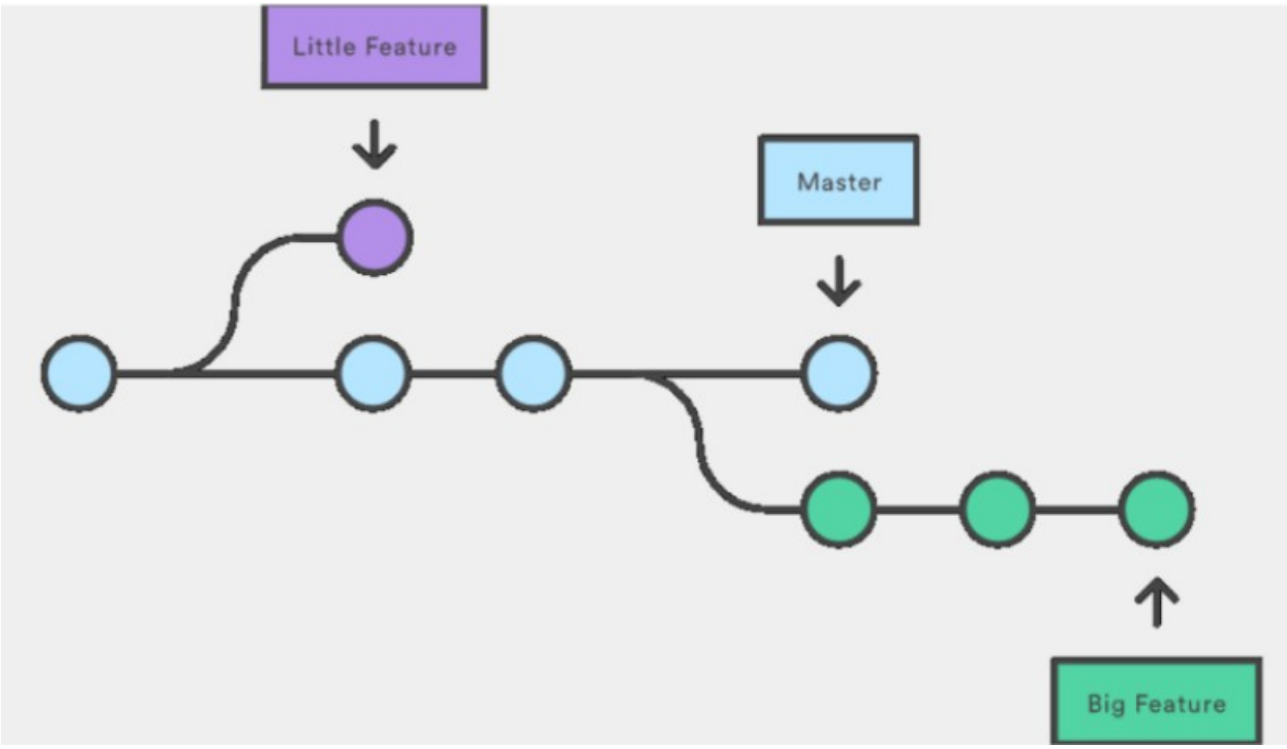
Till now whatever files created and whatever commits we did, all these happend in master branch.
master branch is the default branch/ main branch in git. Generally main source code will be placed in master branch.



master branch devlop web application
branch-1 develop Android compatibility work
branch-2 develop iOS compatibility work

A Branch is nothing but an independent flow of development and by using branching concept, we can create multiple work flows.





Conclusions:

1. Once we creates a branch all files and commits will be inherited from parent branch to child branch.

2. Branching is a logical way of duplicating files and commits. In the child branch we can create new files and we can perform new commits based on our requirements.

3. All branches are isolated to each other. The changes performed in master branch are not visible to the new branch and the changes performed in the new branch are not visible to the master branch.

4. Once the work completed in new branch then we can merge that new brach to the main branch or we can push that branch directly to the remote repository.

To View Branches:

To know all available branches in our local repository, we have to use git branch command.

git branch
* master
* indicates that master is current active branch.

- It will show all branches in our local repository.
- By default we have only one branch: master
- master is the default name provided by GIT.

How to Create a New Branch:

We can create a new branch by using git branch command.
Syntax:
git branch brach_name

Eg:
$ git branch new1branch
It will create a new branch: new1branch
$ git branch
* master
new1branch

How to Switch from one Branch to another Branch?

git checkout brach_name

Short-cut Way to Create a New Branch and switch to that Branch

git checkout -b new2branch

Demo

touch a.txt b.txt c.txt
git add a.txt;git commit -m 'a.txt added'
git add b.txt;git commit -m 'b.txt added'
git add c.txt;git commit -m 'c.txt added'
git status
git log –oneline

git checkout -b test
git status
git log –oneline
touch x.txt y.txt z.txt
git add .;git commit -m 'new files added'
git log –oneline

git checkout master
ls
git log –oneline
touch d.txt
git add d.txt;git commit -m 'd.txt added'

ls
git log –oneline

git checkout test
ls
git log –oneline

In GIT Branching, new directory won't be created and files won't be copied and just HEAD pointer will be changed. Hence to implement branching zero affort is required in GIT.

Important Conclusions:

1. All branches are isolated to each other. The changes performed in master branch are not visible to the new branch and the changes performed in the new branch are not visible to the master branch after branch is created

2. In GIT branching, logical duplication of files will be happend. For every branch, new directory won't be created. But in other version control systems like SVN, if we want to create a branch, first we have to create a new directory and we have to copy all files manually to that directory which is very difficult job and time consuming job.

3. In Git, if we switch from one branch to another branch just HEAD pointer will be moved, beyond that no other work will be happend. Hence implementing branching concept is very easy and very speed.

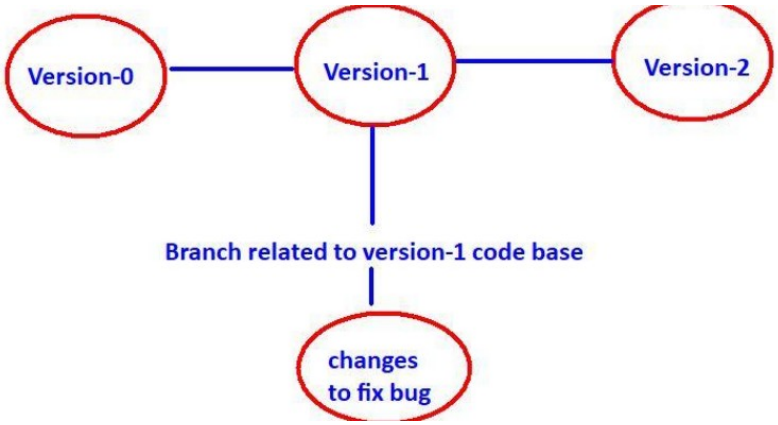
Multiple Use Cases where branching is required:

1. If we are working on a new feature of the project, and if it is required longer time then we can use branching. We can create a

separate branch for Implementing new feature. It won't affect main code (master branch).

2. If we required to work on hot fixes of production code, then we can create branch for the production code base and we can work on that branch. Once work completed then we can push the fixed code to the production. Most of the real time projects have a separate production branch to handle this type of requirements.

To test new technologies or new ideas without effecting main code base, branching is the best choice.

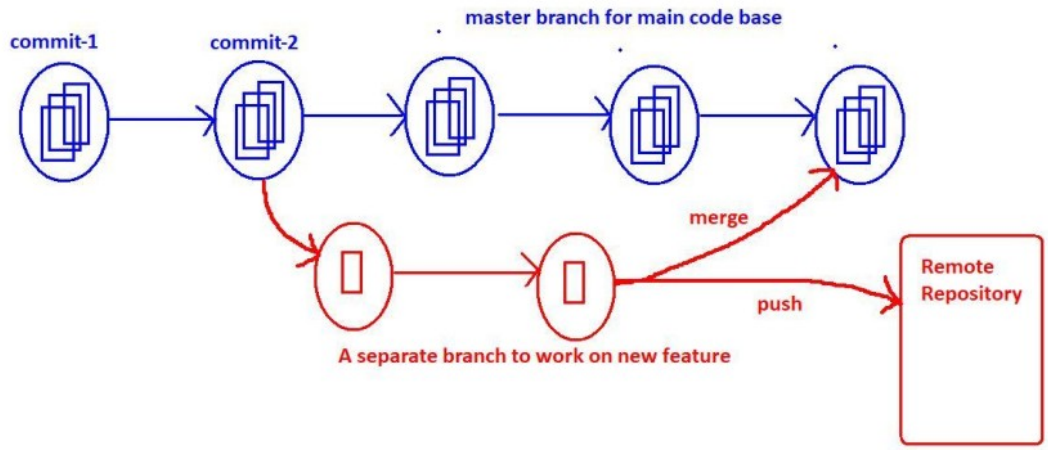


Advantages of Branching:

1. We can enable Parallel development.
2. We can work on multiple flows in isolated way.
3. We can organize source code in clean way.
4. Implementing new features will become easy
5. Bug fixing will become easy.
6. Testing new ideas or new technologies will become easy.

Merging of a Branch:

We created a branch to implement some new feature and we did some new changes in that branch, once work completed we have to merge that branch back to parent branch.



Demo:

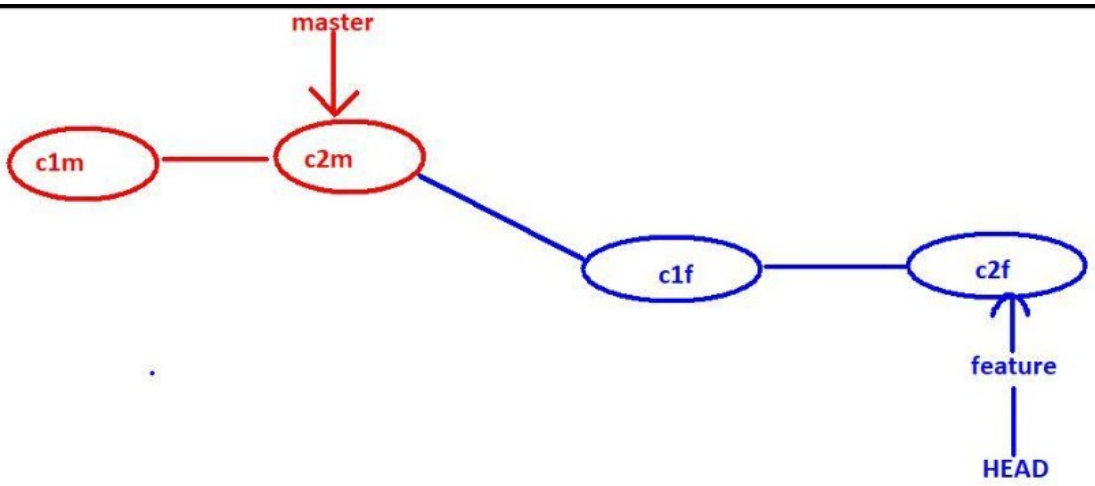
touch a.txt b.txt
git add a.txt; git commit -m 'c1m'
git add b.txt; git commit -m 'c2m'
git log –oneline

git branch

git checkout -b feature

git branch
touch z.txt
git add z.txt;git commit -m 'c1f'
touch x.txt
git add x.txt;git commit -m 'c2f'

git log --oneline master
git log --oneline feature



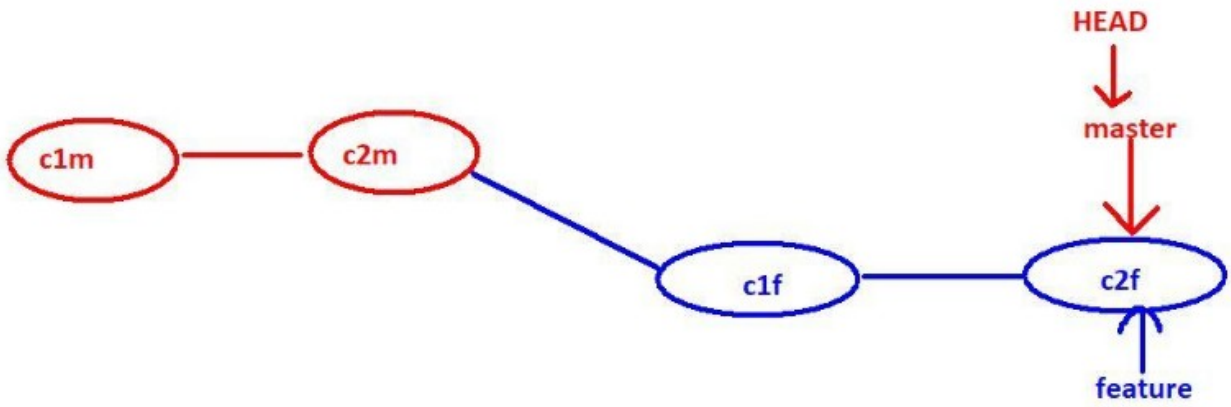
Assume new feature implemented properly, We want to merge feature branch with master branch.

git checkout master
git merge feature
ls
git log –oneline

What is Fast-forward Merge?

After creating child branch, if we are not doing any new commits in the parent branch, then git will perform fast-forward merge. i.e updations(new commits) happened only in child branch but not in parent branch.
In the fast-forward merge, git simply moves parent branch and points to the last commit of the child branch.

No Chance of raising conflicts in fast-forward Merge



What is Three-Way Merge?

If changes present in both parent and child branches and if we are trying to perform merge operation, then git will do three-way merge.

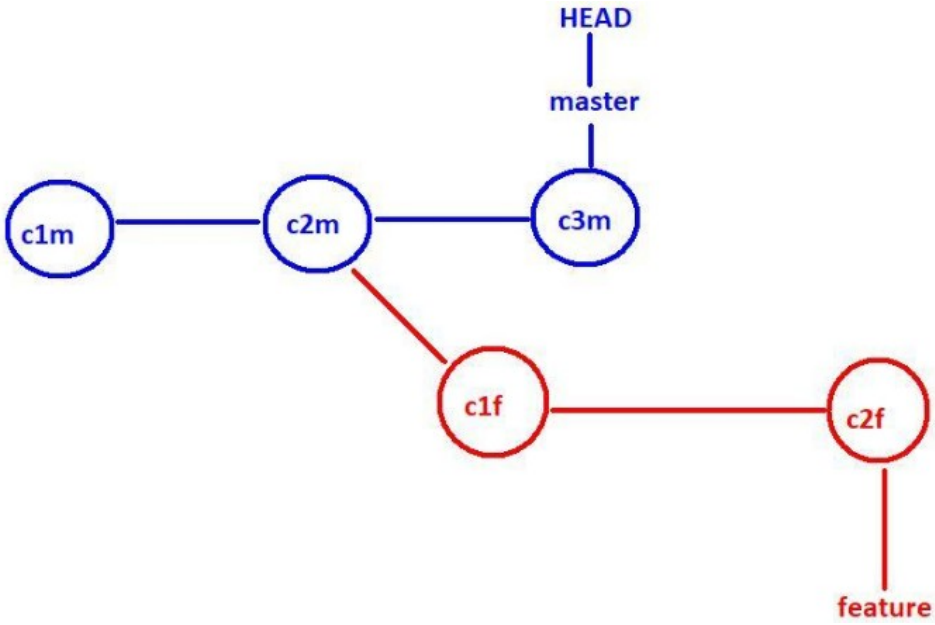
Demo:

touch a.txt b.txt
git add a.txt;git commit -m 'c1m'
git add b.txt;git commit -m 'c2m'
ls
git log –oneline

git checkout -b feature
touch x.txt z.txt
git add x.txt ; git commit -m 'c1f'
git add z.txt ; git commit -m 'c2f'

git branch
git log --oneline master

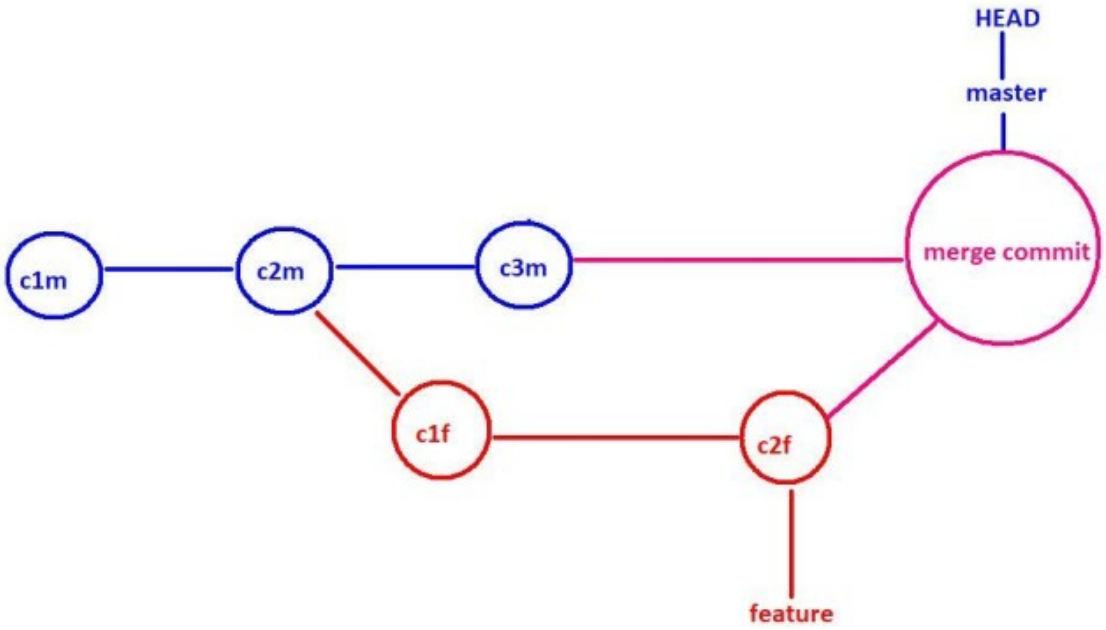
git log --oneline feature



Now we want to add new stuff to master branch after creating feature branch

git checkout master
touch c.txt
git add c.txt ; git commit -m 'c3m'
git merge feature **Merge made by the 'recursive' strategy.**

Three-way merge creates a new commit which is also known as merge commit. Parent branch will pointing to the newly created merge commit.



git log --oneline –graph

Fast-forward	Three-way Merge
1. After creating child branch, if updates are available only in the child branch but not in the parent branch, then GIT will perform Fast-Forward Merge.	1. After creating child branch, if updates are available in both Parent and child branches, then GIT will perform Three-way Merge.
2. It does not require any additional commit	2. It requires a new commit which is also known as Merge commit.
3. There is no chance of conflicts because new commits are available only in child branch but not in parent branch.	3. There may be a chance of conflicts because new commits are available in both parent and child branches.
4. Fast-forward merge is fully handled by GIT.	4. If there is a conflict, we may required to handle manually.

Merge Conflicts and Resolution Process

- In the case of 3-way merge, if the same file updated by both Parent and child branches then may be a chance of merge conflict.
- If there is a conflict then GIT stops the merge process and provides conflict message.
- We have to resolve the conflict manually by editing the file.
- Git will markup both branches content in the file to resolve the conflict very easily.
- Once we completed editing of the file with required final content, then we have to add to the staging area followed by commit.
- With that merging process will be completed.

Demo:

```
mkdir project12
cd project12
git init
echo "First Line Added" > a.txt
git add a.txt ; git commit -m 'c1m'
 echo "Second Line Added" >> a.txt
git add a.txt ; git commit -m 'c2m'

 git checkout -b feature

echo "New Data Added By Feature Branch" >> a.txt
git add a.txt ; git commit -m 'c1f'
```

git checkout master
echo "New Data Added By Master Branch" >> a.txt
git add a.txt ; git commit -m 'c3m'

git diff master feature

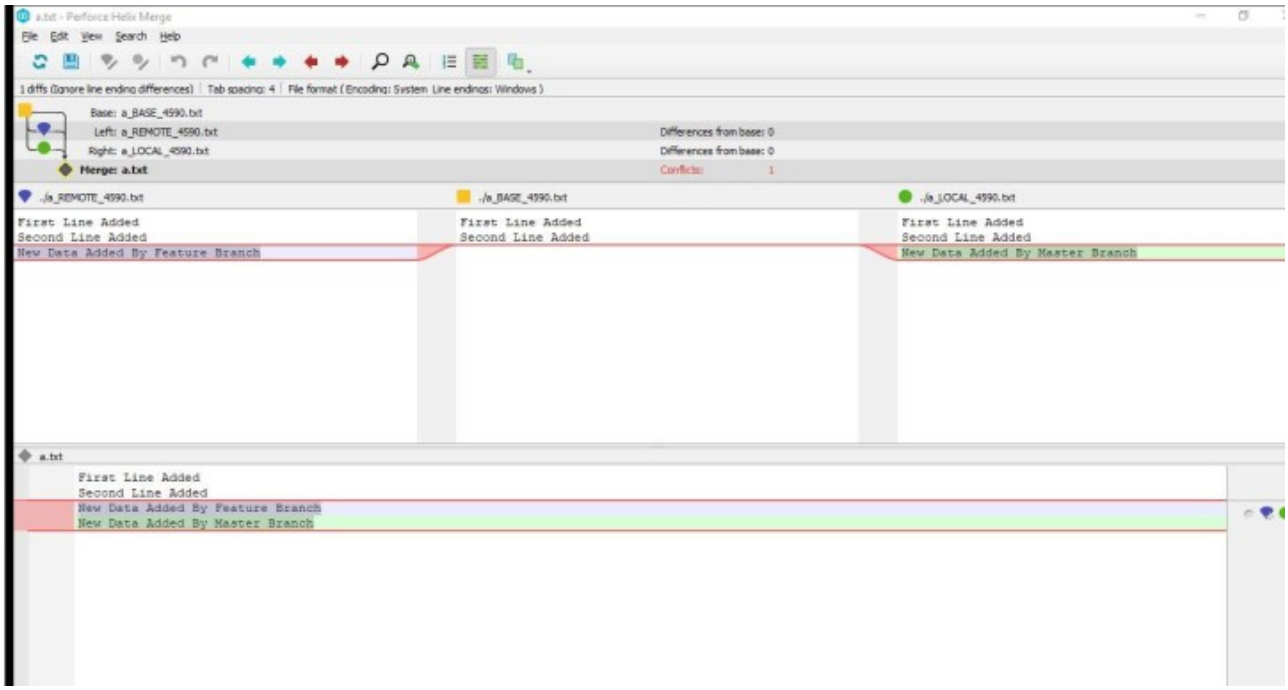
$ git merge feature
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project12 (master|MERGING)

$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")

https://stackoverflow.com/questions/426026/git-on-windows-how-do-you-set-up-a-mergetool

Now how to resolve conflict?

git mergetool



if tool is not working just open file using gedit and update it

finally:
git add a.txt ; git commit -m 'Resolved Merge Conflicts'

git status

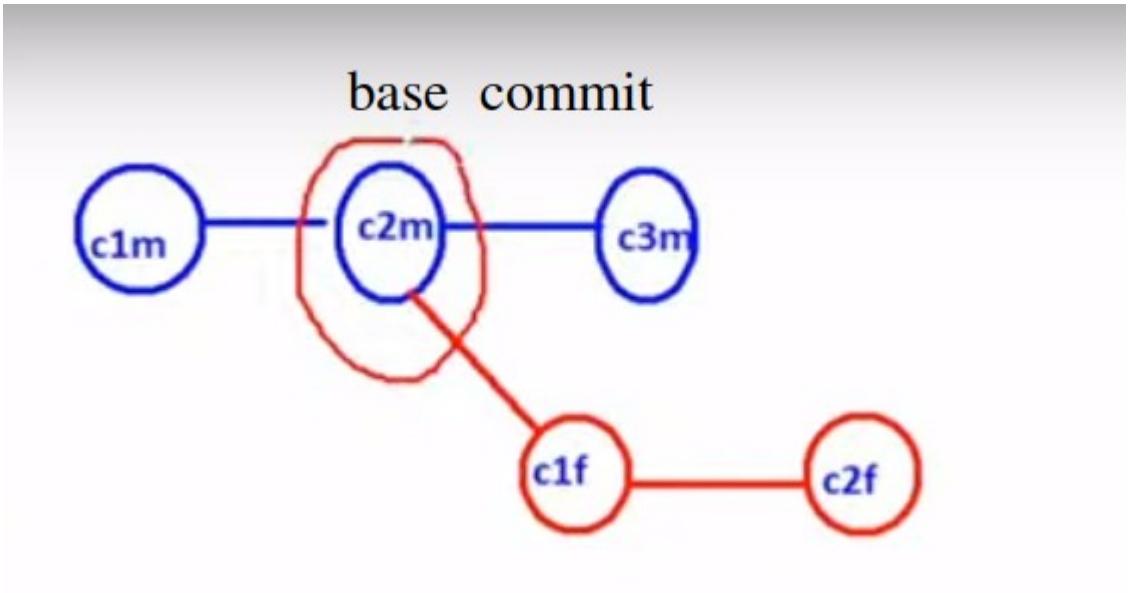
```
raj@raj-Aspire-E5-575G:~/Desktop/devToolsStuff/gitdemo2$ git log --oneline --graph
*   3a91349 (HEAD -> master) Resolved Merge Conflicts
|\
| * 32cd02c (feature) c1f
* | 3bc3a59 c3m
|/
* bf05dc4 c2m
* c19d217 c1m
```

How to Delete a Branch?

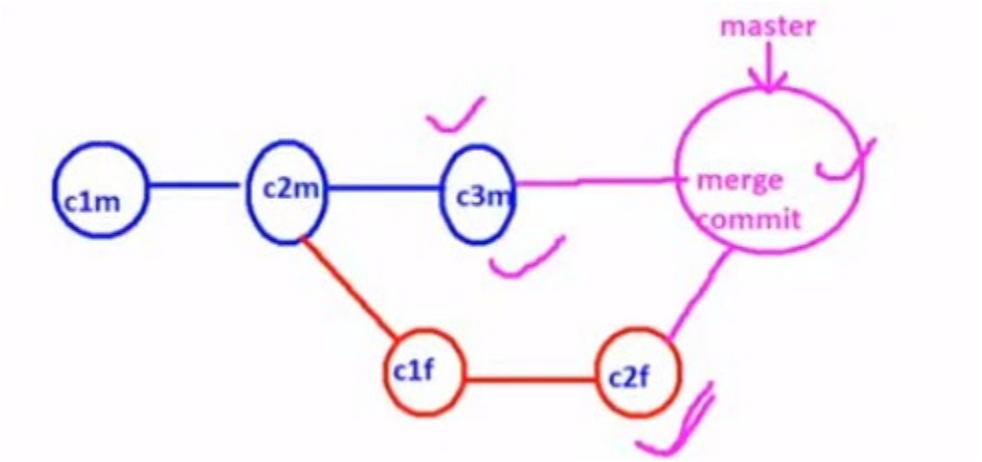
1. Once we completed our work we can delete the branch.
2. Deletion of the branch is optional.
3. The main objective of deleting branch is to keep our repository clean.
4. We can delete a branch by using git branch command with -d option.

Command:
git branch -d feature

Rebase :



We already now that if there changes in master branch after branches then we need 3 way merge



As
graph is non-liner we know which changes are comming from master and which are coming from child branch. Form merge a merge commit is created. If both branch modifify same file in we have conflict that we need to resovle it

What if we want liner graph (for easy understanding) , new commit should not be create (as in previous case)
and we should not have any conflict.

REBASE:
1. liner flow
2. new commit is is not created as in case merge commit
3. we should not have any conflict

REBASE IS ALTERNATIVE TO MERGE OPERATION

BEASE: Re+ base: re arrage the base

process of rebase:(Re arrange base)
Rebase is two way process:
Step 1. we have to rebase feature branch on the top of master branch
Step 2. we have to mege feature branch into master branch (internally fast forward meger)

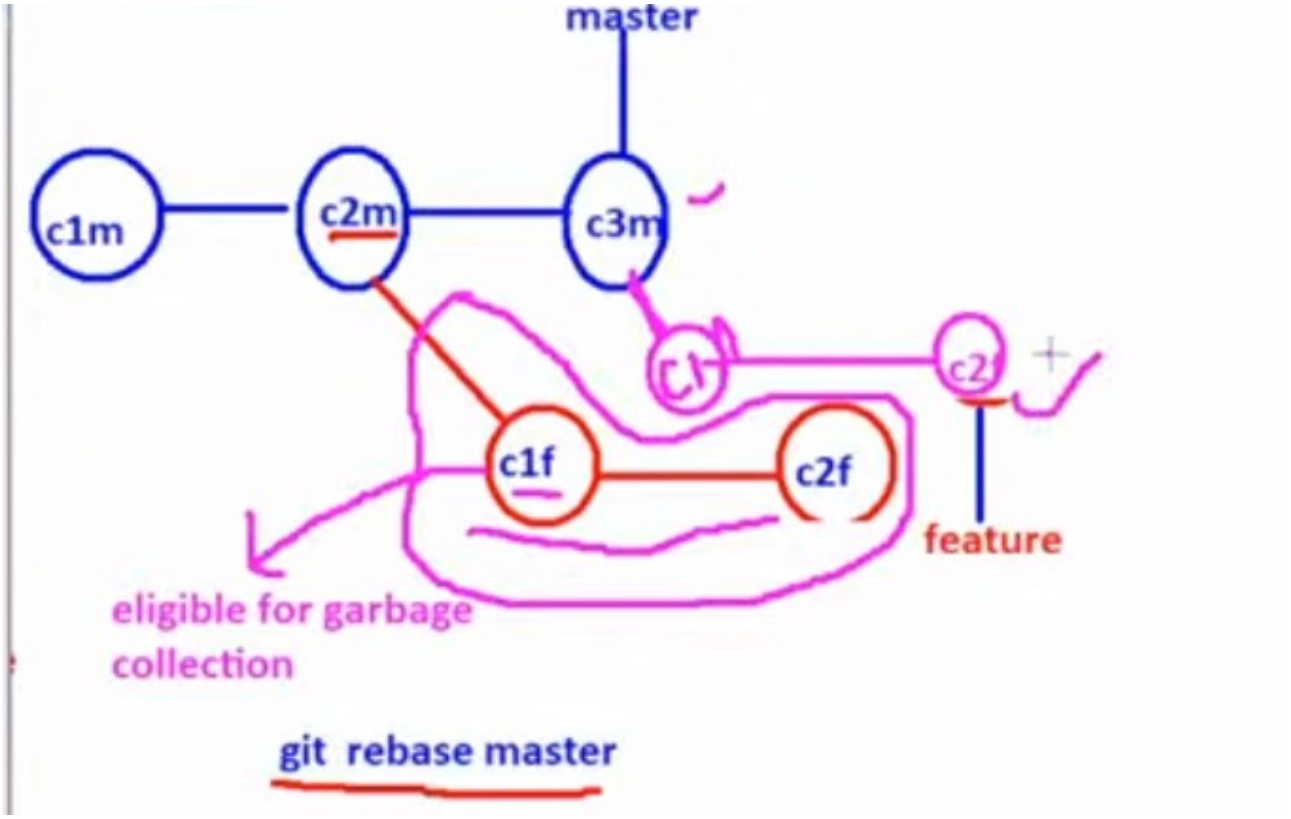
Step 1. we have to rebase feature branch on the top of master branch

git checkout feature

git rebase master

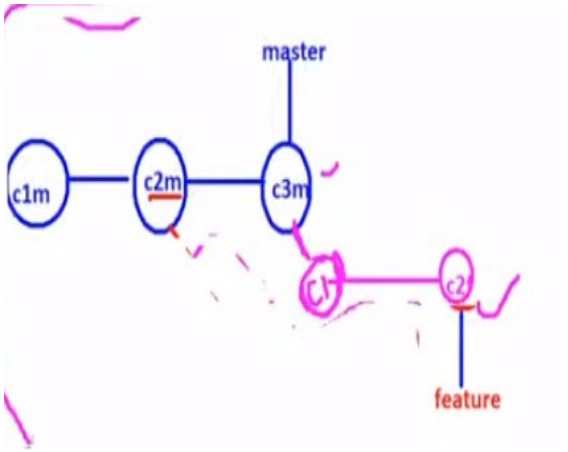
Whenever new commit are there in the feature branch will be duplicated by the git, Here everything would be same (like commit messages, timestemp, author,name email id etc) except commit ids

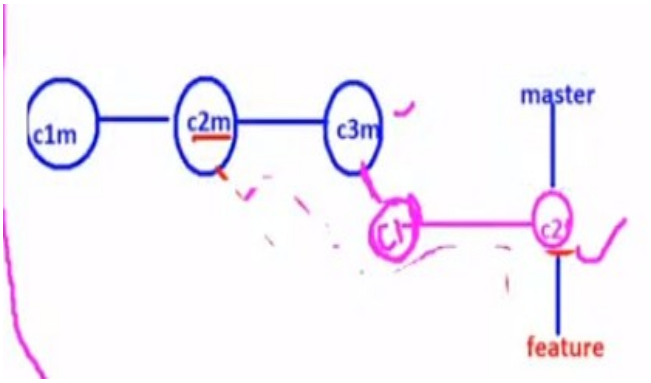
Then base commit of feature branch is updated as the last commit of the parent branch



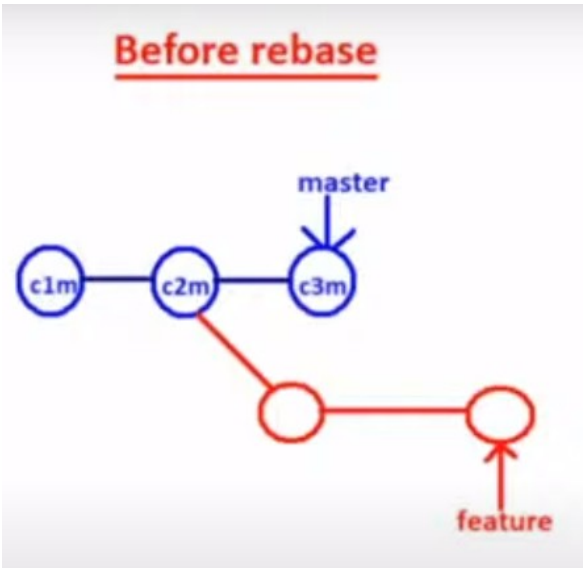
Step 2. we have to mege feature branch into master branch (internally fast forward meger)

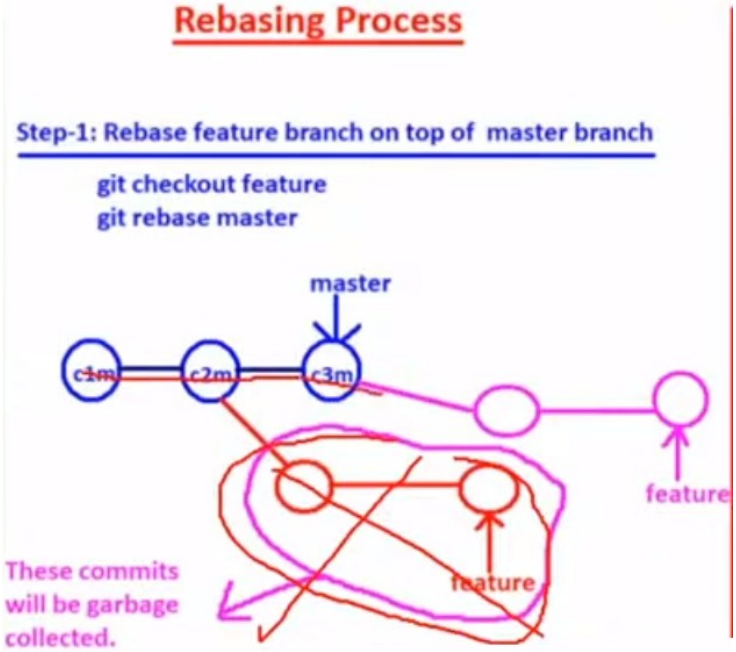
Now if there is no changes in master after branching then git apply **fast forward merge**

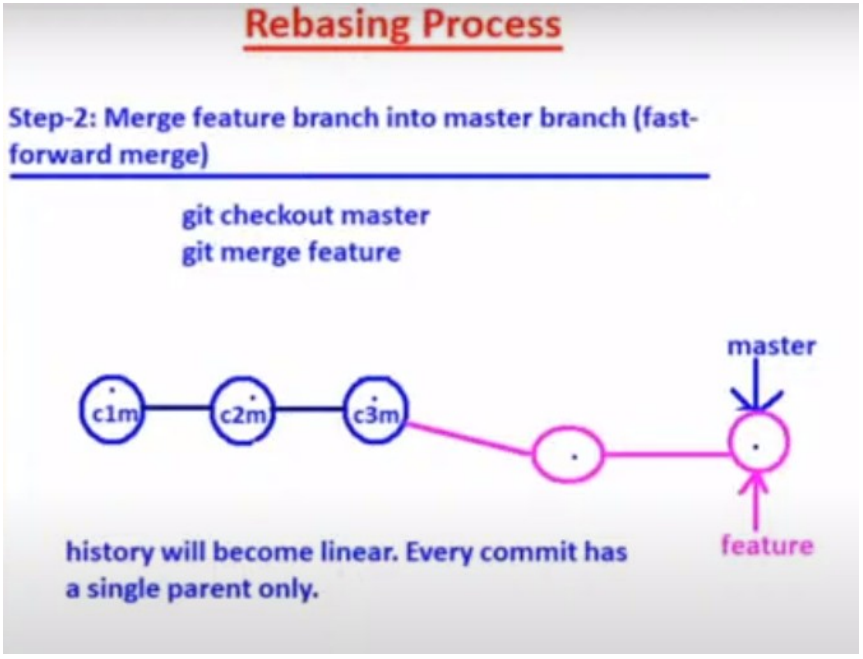




git checkout master
git merge feature
No extra merge is created and no chance of conflict and we will get liner flow, easy to understand for developer







Example:

touch a.txt b.txt
git add a.txt;git commit -m 'c1m'
git add b.txt;git commit -m 'c2m'
ls
git log --oneline

git checkout -b feature
touch x.txt z.txt
git add x.txt ; git commit -m 'c1f'
git add z.txt ; git commit -m 'c2f'

git branch
git log --oneline master

git log --oneline feature

git checkout master
touch c.txt
git add c.txt ; git commit -m 'c3m'

Now apply step 1:
git checkout feature
git rebase master

Now apply step 2:
git checkout master
git merge feature

git log master --oneline --graph
* 9a6a7d5 (HEAD -> master, feature) c2f
* e352929 c1f
* f83ab08 c3m
* 8119cab c2m

* cbad643 c1m

Advantage:

1. Rebase keep history clean, every commit have a single parent
2. clear workflow (liner) will be there, easy for developer to understand
3. use fast forward merge no chances for conflict
4. No extra merge commit is created

Disadvantage

1. Which steps are comming from child branch which are commit is commit from master (as histroy is re-written). We are not aware which changes comming from child branch
2. On public repository never recommented to use rebase as it rewrite history, must not be used.

merge	rebase
1. It is a single step process. git checkout master git merge feature	1. It is a 2-step process git checkout feature git rebase master git checkout master git merge feature
2. merge preserves history of all commits.	rebase clears history of feature branch
3. The commits can have more than one parent and hence history is non-linear	3. Every commit has a single parent. hence history is linear
4. conflicts	4. no chance of conflicts
5. we can aware which changes are coming from feature branch	we cannot aware which changes are coming from feature branch
6. we can use merge on public repositories	6. public repositories

If we want to combine all commits of feature branch into a single commit and merge that
commit to the master branch, then we should go for squash option.

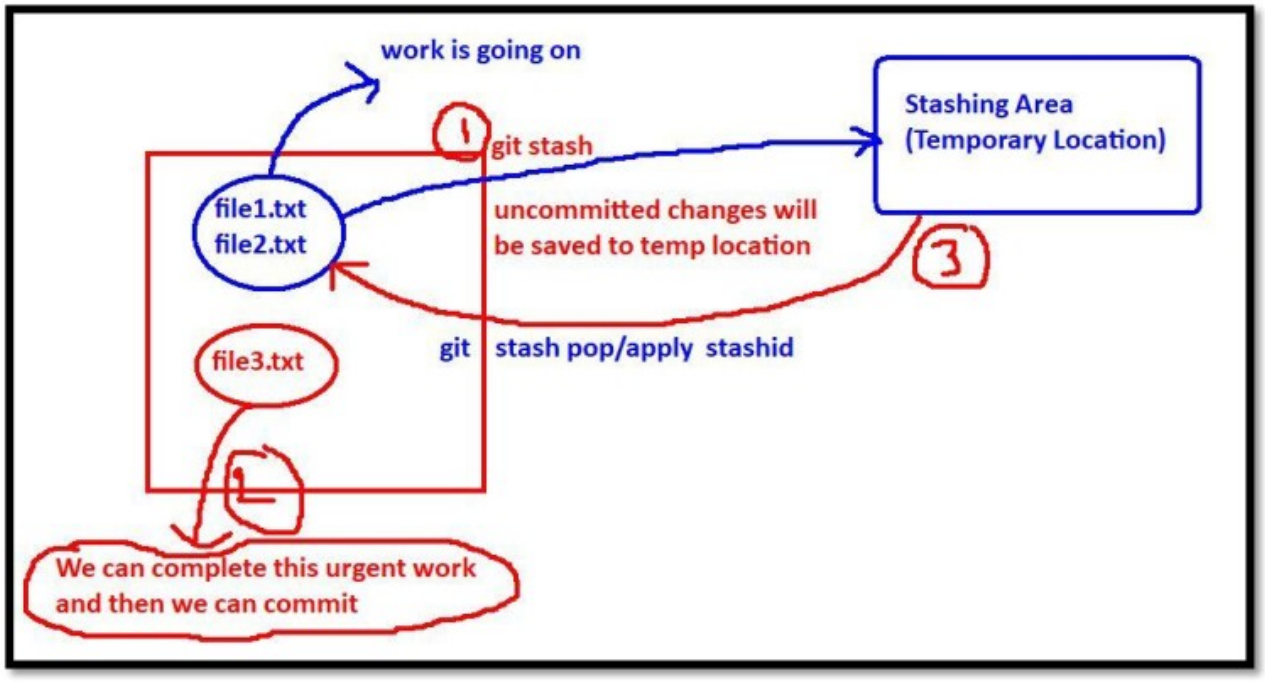
git merge --squash feature

Stash in GIT

something put away or hidden: a stash of gold coins buried in the garden. a place in which something is stored secretly; hiding place; cache.

What is git stash:

The git stash command takes our uncommitted changes (both staged and unstaged), saves in some temporary location. After completing our urgent work, we can bring these stashed changes to our current working directory



1. Stashing concept is applicable only for tracked files but not for newly created files.
2. To perform stashing, atleast one commit must be completed.

Demo
mkdir stashing cd stashing
git init

echo "First Line in File1" > file1.txt
echo "First Line in File2" > file2.txt
git add file1.txt file2.txt;git commit -m '2 files added'
vim file1.txt (add some data)
vim file2.txt

git add file2.txt
git add file1.txt

git status
On branch master
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
modified: file2.txt
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified: file1.txt

Assume we required to create and work on file3.txt and and this file changes needs to be committed immediately.
To work on file3.txt, we have to save uncommitted changes of file1.txt and file2.txt to some temporary location, because we don't want to include these changes in the current commit.

For this we should go for git stash command.

git stash
Saved working directory and index state WIP on master: 0323e16 2 files added

git status
On branch master
nothing to commit, working tree clean
lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/stashing (master)

How to list all available stashes:
git stash list
stash@{0}: WIP on master: 0323e16 2 files added

How to check the contents of stash:
git show stash@{0}

How to perform unstash
git stash pop stash@{0}
git stash apply stash@{0}

to unstash all:
git stash pop

How to clear stash
git stash clear

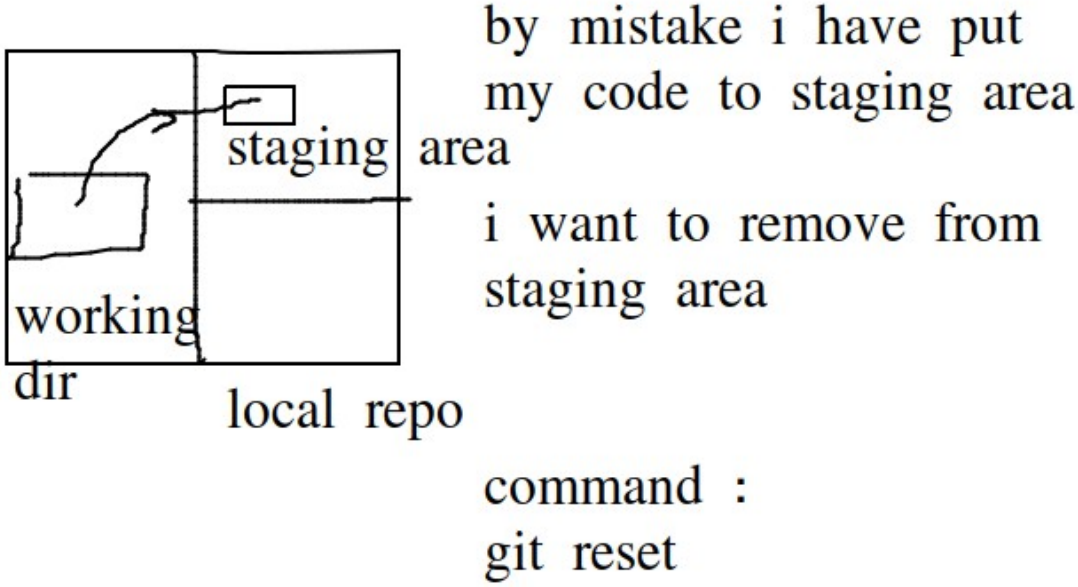
git reset command

git reset is a powerful command used to undo local changes to the state of git repo

Three type: soft, mixed and hard

Reverse of git add .

To seet the changes from both staging area and working directory at a tie
git rest –hard



git revert command

The revert command helps you to undo an existing commit

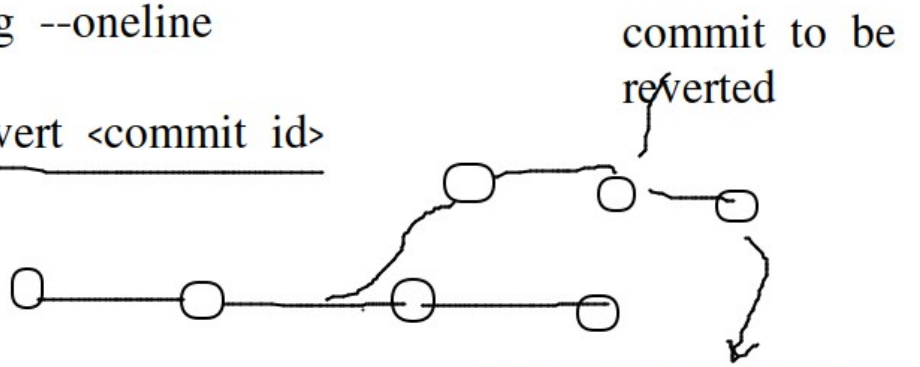
it does not delete anyh data in the repo rather git create a new commit with the included file revert to their previous state so, your version control history move forward while the state of your file moves backword

reset => before commit
revert => after commit

```
mkdir demo
cd demo
cat > newfile
    hi this is a sample file

git add .
git commit -m "first commit"
git log --oneline

git revert <commit id>
```



How to remove untrached files

git clean -n (dry run : git will ask you)
git clean -f (forcefullly)

Tags

Tag operation allows giving meaningful name to a specific verion of repository

to apply tags:
git tag -a <tagname> -m <message> <commit id>

to see list of tags:
git tag

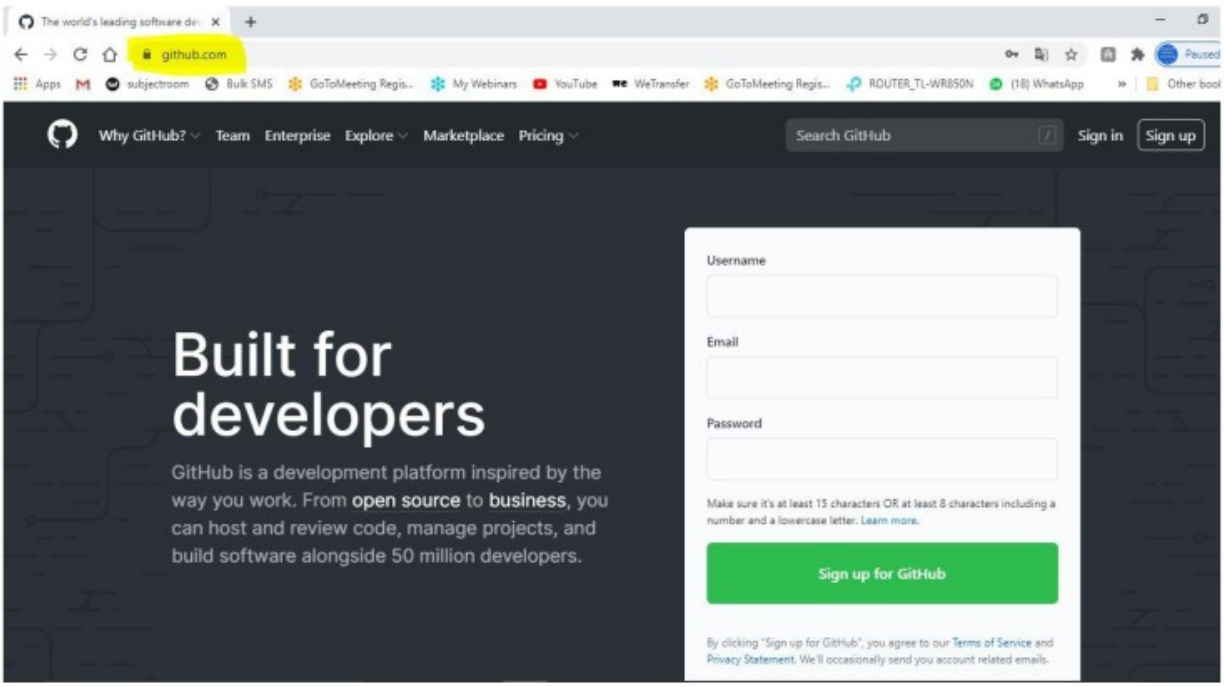
to see particular commit contents by using tag
git show <tag name>

to delete a tag
git tag -d <tag-name>

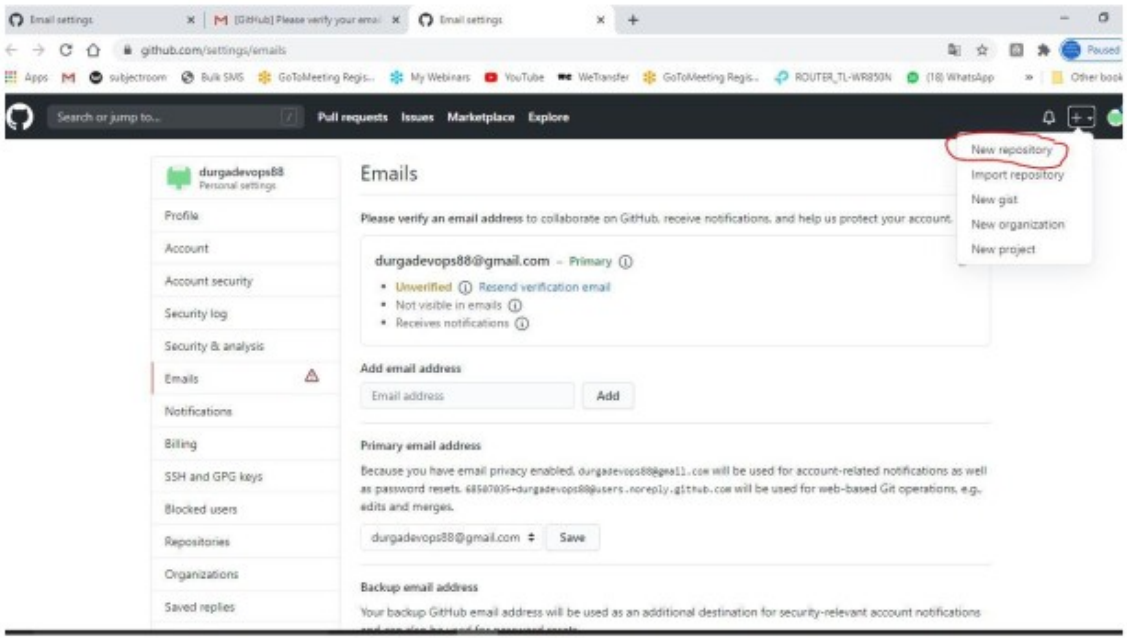
Working with Remote Repositories

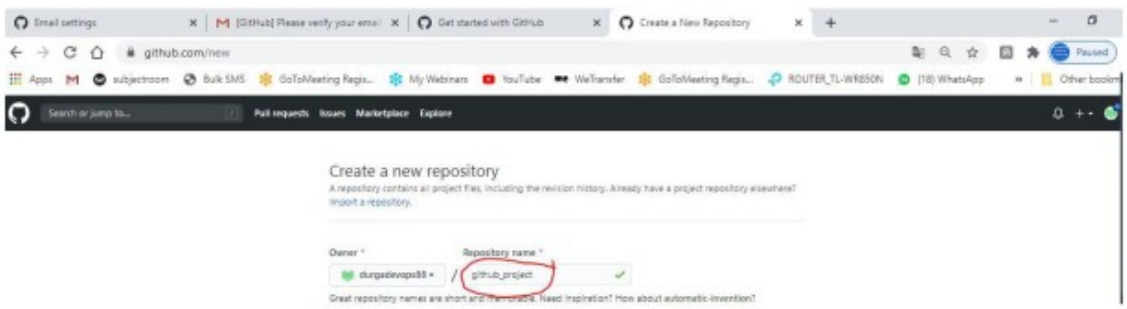
Github is an online repository store/remote repository service provider. We can store our repositories in github so that our code is available for the remaining team members.

https://github.com



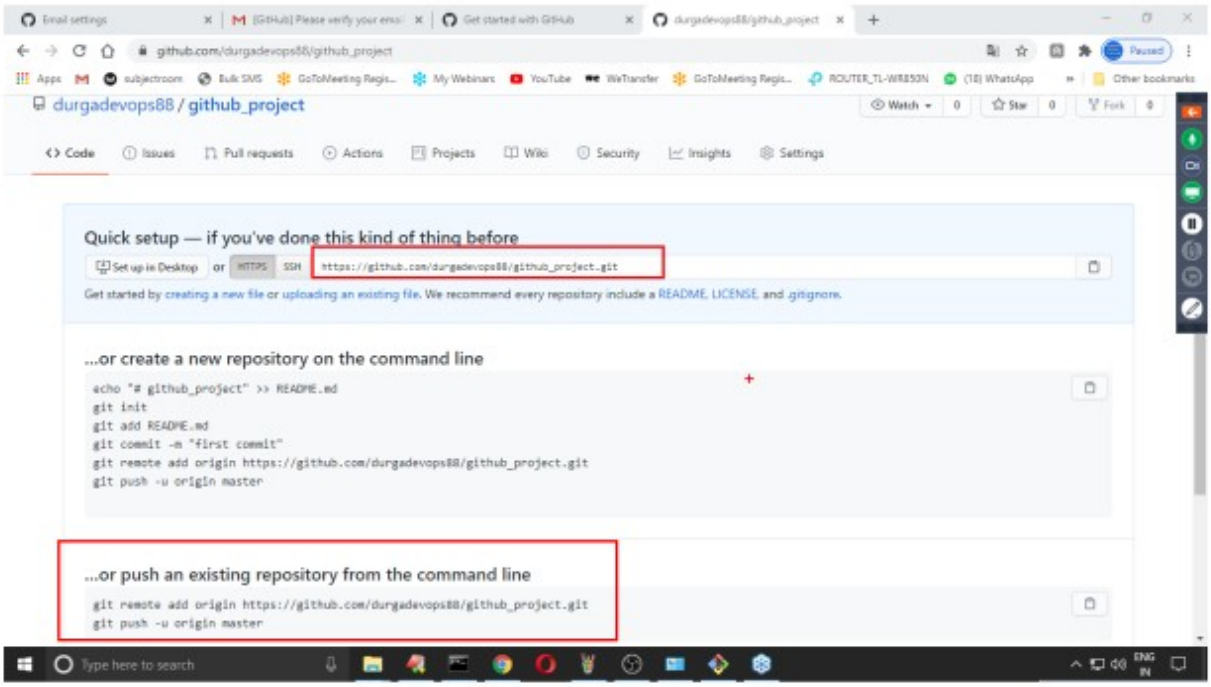
e to





Creating repo with github

Repository Name is the name of the project folder. We have to choose meaningfull name.



How to work with Remote Repository:

To work with remote repository we have to use the following commands

1. git remote
2. git push
3. git clone
4. git pull
5. git fetch

git remote

T configure remote repository to our local repository.

git remote add <alias_name> remote_repository_url

Eg:

$ git remote add origin https://github.com/rgupta00/github_project.git

Here, origin is alias name of the remote_repository_url. By using this alias name only we can communicate with remote repository.

Instead of origin we can use any name, but it is convention to use origin.

By using git remote command we can also view remote repository information.

git remote
It just provides the alias names of remote repositories

git remote -v
It provides remote repository urls also.

git push
We can use git push command to send our changes from local repository to remote repository. ie to push our changes from local repo to remote repo.

git push <remote_name> <branch_name>
git push origin master

git clone
cloning means creating exactly duplicate copy.

Already there is a project on remote repository. Being a new team member, we may required complete remote repository into our local repository. For this purpose we have to use git clone command.

i.e we can use git clone command to create local repository with remote repository files. All the files and commit history will be copied from remote repository into local repository.

Syntax: git clone <remote_repo_url>

git clone https://github.com/rgupta00/github_project.git

Now the project name will become repository project name.

Note: Based on our requirement we can provide a new name for the project.
git clone <remote_repo_url> <new_project_name>
git clone https://github.com/rgupta00/github_project.git my_project

Q. Before using git clone command is it required to use git init command?
Ans: No, git clone command itself will create local repository and hence we are not required to use git init command explicitly.
Q In how many ways we can create a new local repository?
Ans: In 2 ways
1. By using git init command to create empty local repository.
2. By using git clone command to create local repository with the files of remote repository

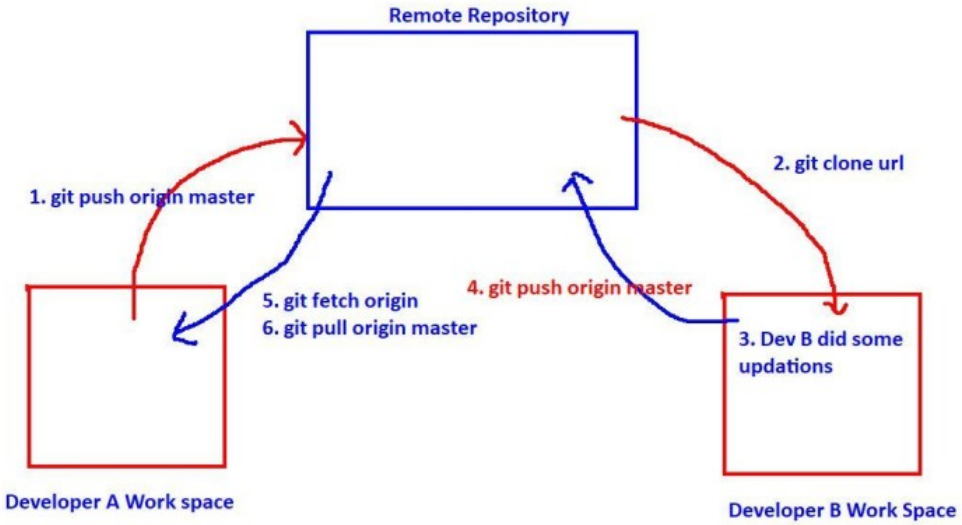
git fetch Command

We can use get fetch command to check whether any updates available at remote repository or not. This command will retrieve only latest meta-data info from the remote repository.
It won't download updates from remote repository into local repository.

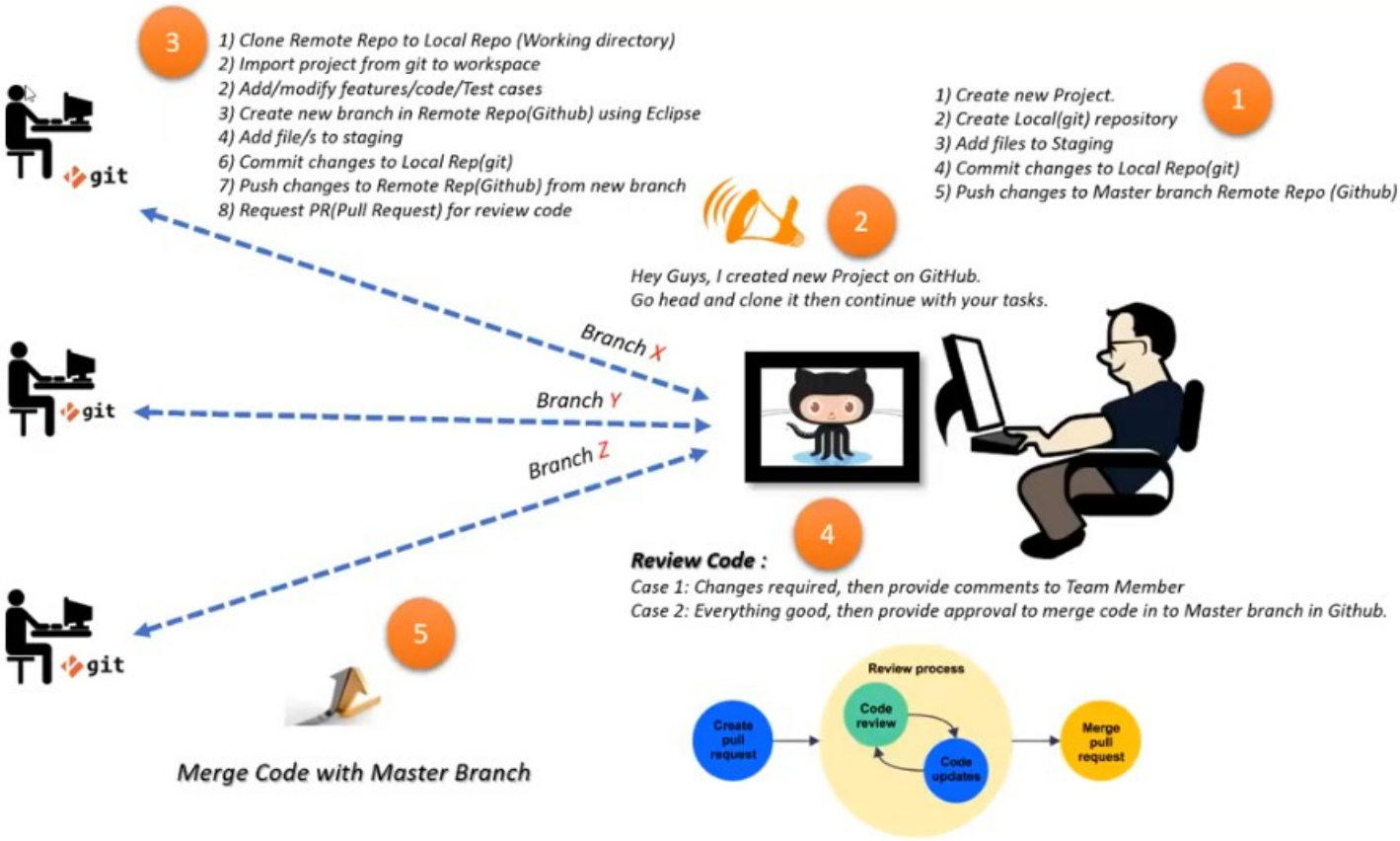
git fetch origin

git pull Command

We can use git pull command to download and merge updates from remote repository into local repository.
git pull = fetch+merge
Syntax:
$ git pull <remote> <branch>
$ git pull origin master



GIT ECLIPSE



https://stackoverflow.com/questions/18813847/cannot-open-git-upload-pack-error-in-eclipse-when-cloning-or-pushing-git-repos