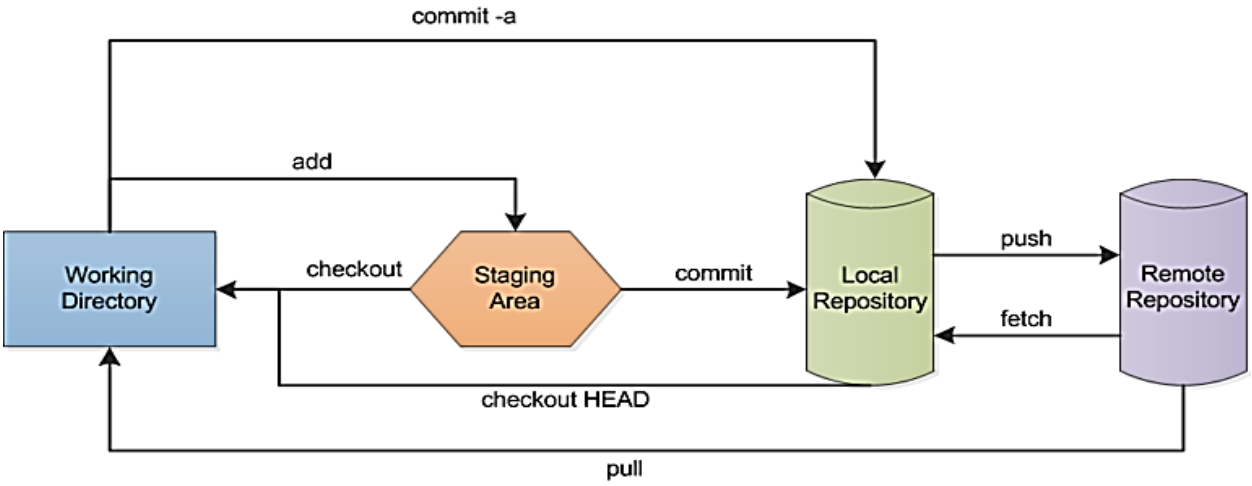
GIT Complete Notes

GIT is a three phased commit. Git is a Version controlled system. Git follows distributed system.
It has Four stages
1. Working Tree: Working Tree is a working folder where code is written by an individual.
It is present in my work folder
> Git Add is used for sending to next level(staging).
2. Staging Area / Indexing area: Staging Area is the place where changes have to be added
> Git Commit is used for sending to next level (Local Repo)
After staging area is completed then it is committed to Local Repo
*** While committing it consider three factors ***
a) What is the time (Ex: default time of system)
b) Who is committing (Ex: $ git config --global user. Email xxxxx $ *git config --global user.name xxxx*)
c) What are the changes with commit message (EX: git commit -m "<commit message>")
3. Local Repo
Process diagram:
Working Tree ---**ADD + File Name**---> Staging Area ---**COMMIT + Message**--> Local Repo
4. Remote Area
> Git Push origin master to push to remote area for master branch



commit -a: Directly commit modified and deleted files into the local repository (*no new files!*)
add: Add a file to the staging area.
checkout: Get a file from the staging area.
checkout HEAD: Get a file from the local repository
commit: Commit files from the staging area to the local repository
push: Send files to the remote repository
fetch: Get files from the remote repository
pull: Get files from the remote repository and put a copy in the working directory

Note

- Git never give importance to empty folders. Git stores data in **SHA-256 Hash algorithm**. For same data same Hask id is given. Git never do anything to untracked files in branching concept
- Repositories created with git init --bare are called bare repos. They are structured a bit differently from working directories.
  - bare repos store git revision history of your repo in the root folder of your repository instead of in a. git subfolder.

Git Work Setup

*******Create Repo********

$ ***git init***

---> to initialize git repository on local machine. After that We can see a folder. git on local machine.

*******Syncing Repo*******

$ ***git remote add origin*** *"<Link of remote Repo>"*

Ex: git remote add origin "https://github.com/crsreddy1447/RAJA.git"

we can get this link after signup to GitHub click on clone.

$ ***git push origin master*** -----> To send all the files from local repo to central repo

$ ***git pull origin master*** -----> To fetch all the files from central repo to local repo.

********Making Changes**********

To access to GitHub use below 2 steps

$ ***git config --global user.email*** *"crsreddy1447@gmail.com"*

$ ***git config --global user.name*** *"crsreddy1447"*

*****************How to Add GITHUB REPO and PUSH from CLI****************************

$ ***git remote add origin git@github.com:****<Username>/<Repo Name>.git*

Ex: git remote add origin git@github.com:crsreddy1447/MBA_Project.git

$ ***git push -u origin master***

$ ***git remote -v*** # -v. Shows URLs of remote repositories when listing your current remote connections

$ ***git pull*** *<remote>*

origin git@github.com:crsreddy1447/MBA_Project.git (fetch)

origin git@github.com:crsreddy1447/MBA_Project.git (push)

******************* **Commonly Used Commands** ************************

$ ***git status*** #Tell you which files are added to index and are ready to commit.

$ ***git add*** #This will add files to your index.

$ ***git add -A*** #To add all the files to staging area

$ ***git add -u*** #To add modified files

$ ***git commit*** #Will commit to local repo

Ex: git commit -m "<commit message>"

$ ***git commit -a -m*** *"<commit message> "#*To commit by adding files to local repo.

$ ***git log*** #It will show entire logs of files and commit with message.

$ ***git reset --hard*** ---> It rests the modified work done in working tree and staging area

$ ***git clean -fd*** *< Dir Path>* ---> to delete or remove a newly created file

$ ***git checkout*** *--<file Path>* ---> to reset a specific file in working tree

$ ***git log --oneline*** ---> to get short commit IDs

$ ***git checkout*** *<commit id>* ---> to move across history or commits

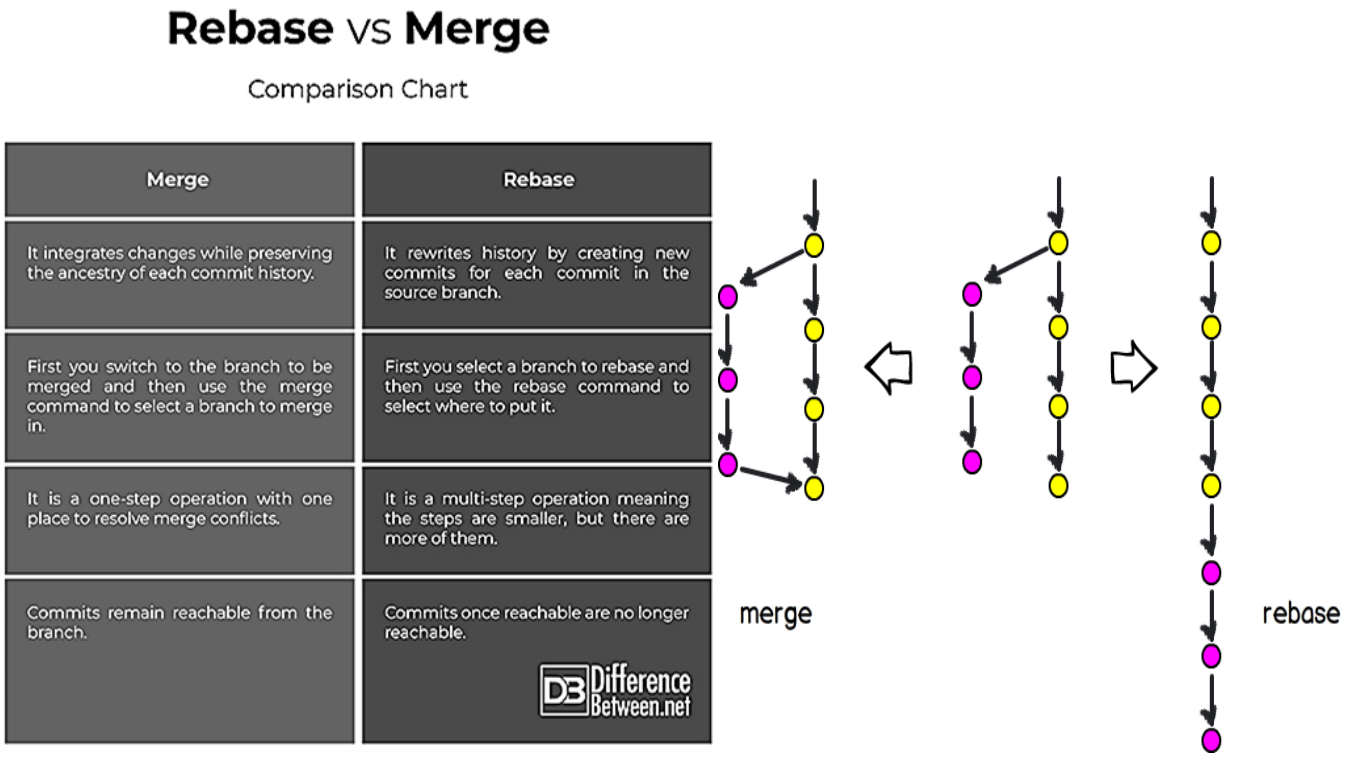
$ ***git cat-file -p*** *<full commit id>* ---> to see the content in that commit.

$ ***cat .gitignore*** ---> file specifies intentionally untracked files that Git should ignore

******<u>**Parallel Development**</u>*****
It helps in creating branches and combine code to master
$ ***git branch <branch name>*** #This will create a new branch
$ ***git checkout <branch name or master>*** #switch to other branch or master
Ex: git checkout master #switches to master
git checkout firstbranch #switches to first branch.

********** <u>**MERGE VS REBASE**</u> **********
<u>**Merging:**</u> Used to merge branches. While doing merge you have to be in target branch.
$ **git merge** <name of the branch that has to be merged with master>
Ex: git merge firstbranch # Now we can see entire content of firstbranch in master
*Note: It integrates the changes made in different branches into one single branch

<u>**Rebasing:**</u> Similar to merge. But it points on to tip of master in a linear way.
$ ***git rebase*** <from which branch the data have to merge>
Ex: (master branch) $ git rebase firstbranch ----> it merges the data from firstbranch to master
*Note: it is used when changes made in one branch needs to be reflected in another branch
$ ***git rebase -i HEAD~<Position from top>*** #To modify/delete the commit which is not pushed to Remote Repo
EX: git rebase -i HEAD~2
*Note: This should not done if I it is committed to Remote Repo



Cherry PICK

Sometimes you don't want to merge a whole branch into another, and only need to pick one or two specific commits. To pick some changes into your main project branch from other branches is called **cherry-picking**.
From new-features branch run
$ ***git log --oneline*** #To get a better log of your commit's history.
Checkout the branch where you want to cherry pick the specific commits. In this case master branch:
$ ***git checkout master***
Now we can cherry pick from new-features branch:
$ ***git cherry-pick d467740***
This will cherry pick the commit with hash d467740 and add it as a new commit on the master branch.
Note: it will have a new (and different) commit ID in the master branch.
***If you want to cherry pick more than one commit in one go, you can add their commit IDs
$ ***git cherry-pick d467740 de906d4***
***If the cherry picking gets halted because of conflicts, resolve them and
$ ***git cherry-pick --continue***

Some scenarios in which you can cherry-pick:
Accidently make a commit in a wrong branch. Then we check the commit id. If we want to commit back to correct branch, then checkout to correct branch and do cherry-pick with commit id. Then the commit look like it is made in master. *** **But mostly Cherry-pick is not preferable**.

STASHING:

Sometimes you want to switch the branches, but you are working on an incomplete part of your current project. You don't want to make a commit of half-done work. Git stashing allows you to do so. The git stash command enables you to switch branches without committing the current branch.
$ ***git stash***
$ ***git stash list***
$ ***git stash apply stash@{0}*** -->Apply one of the older stashes, you can specify it by naming it
$ ***git stash pop*** # Command will re-apply the previous commits to the repository
$ ***git stash drop*** # Command is used to delete a stash from the queue

******* **GIT Remote Connection******
Connecting Local repo with remote repo
1. SSH connection
2. Generate keygen
3. open key file
4. go to setting ---> ssh and GPG keys ---> new ssh key
5. paste the key
6. connect using $ ssh -T git@github.com
$ ssh-keygen
$ cat /c/Users/Rajashekar/.ssh/id_rsa.pub
$ ssh -T git@github.com

********** **PULL vs FETCH** **********
How to Git pull vs Git fetch
Git clone a specific branch
$ git clone -b <branch> <remote repo>
Ex: $ git clone -b my-branch crsreddy1447/myproject.git
(OR)
$ git clone --single-branch --branch <branch name> <remote-repo>
Ex: git clone --single-branch -b dev crsreddy1447/myproject.git

GIT FETCH VERSUS GIT PULL

Git Fetch	Git Pull
Command: git fetch <remote>	**Command**: git pull <remote> <branch>
Fetch retrieves all the changes from the remote repository without integrating them with the local repository.	Pull grabs the remote copy of the branch and merge the changes with the local repository. Pull is basically fetch followed by the merge command.
It updates the repository data leaving your local repository unchanged.	It updates your local repository with the changes from the remote repository.
It imports commits to local branches to keep you up-to-date with what everybody is working on.	It brings a local branch up-to-date with the remote copy while also updating the other remote-tracking branches.
No merging means no conflicts. They usually occur at the latter stage.	Merge conflicts can occur with git pull when two people are working on the same piece of code.

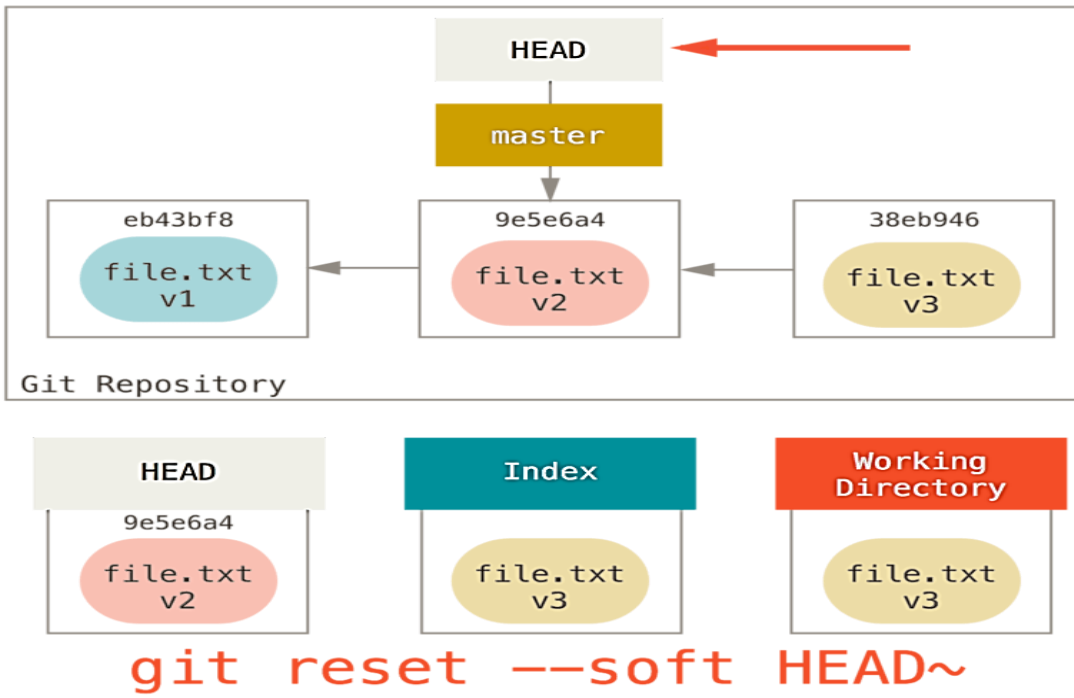
DifferenceBetween.net

GIT RESET. TYPES OF GIT RESET

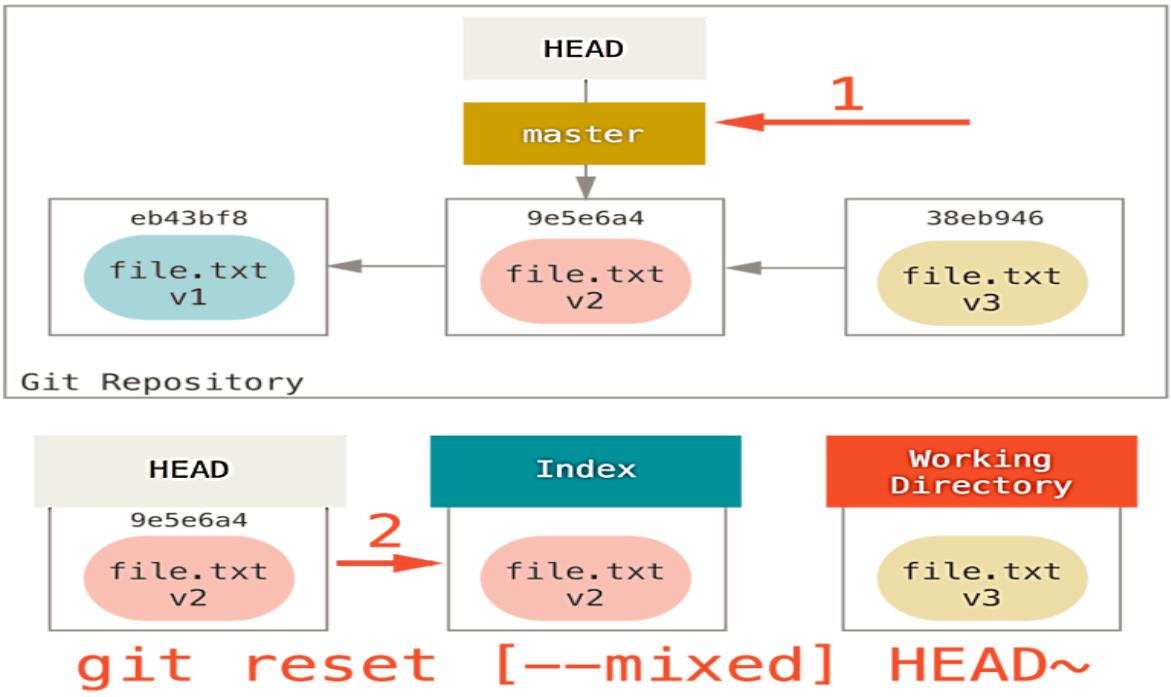
Three types

1. **Soft (Move HEAD):** Here the Head move to previous commit and index and working directory are same.

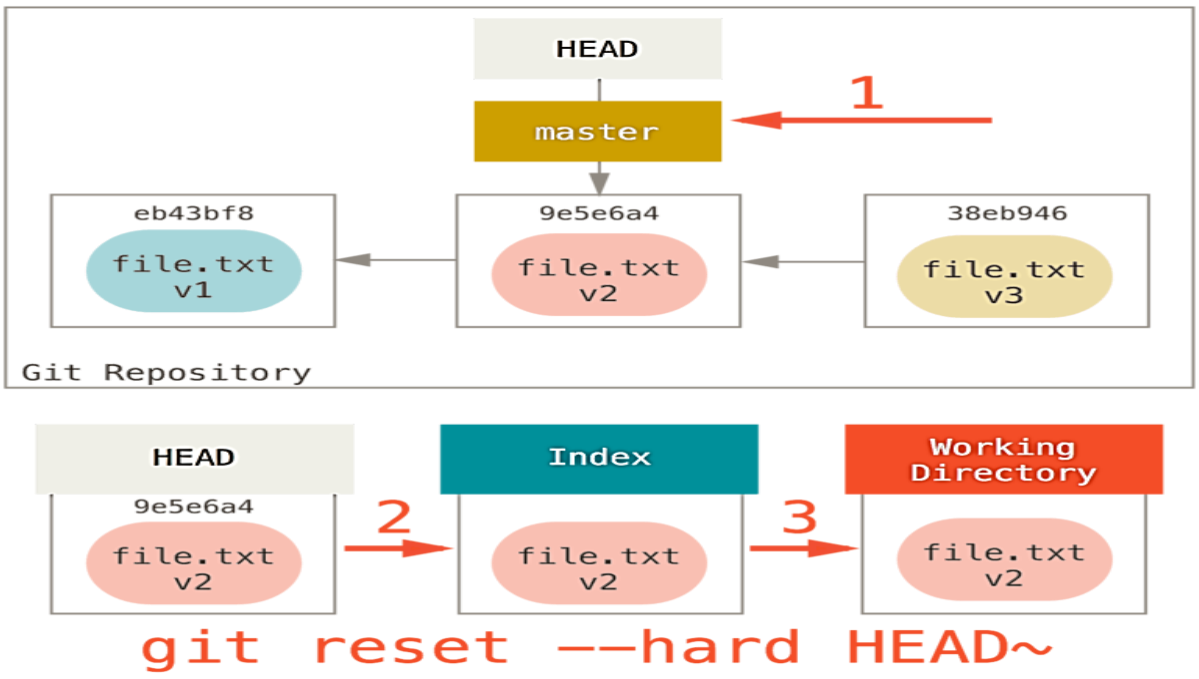
$ ***git reset --soft Head~***



2. Mixed (Updating the Index): Here Head and index file changes (also unstaged). But working dir. is same
$ ***git reset --mixed HEAD~***

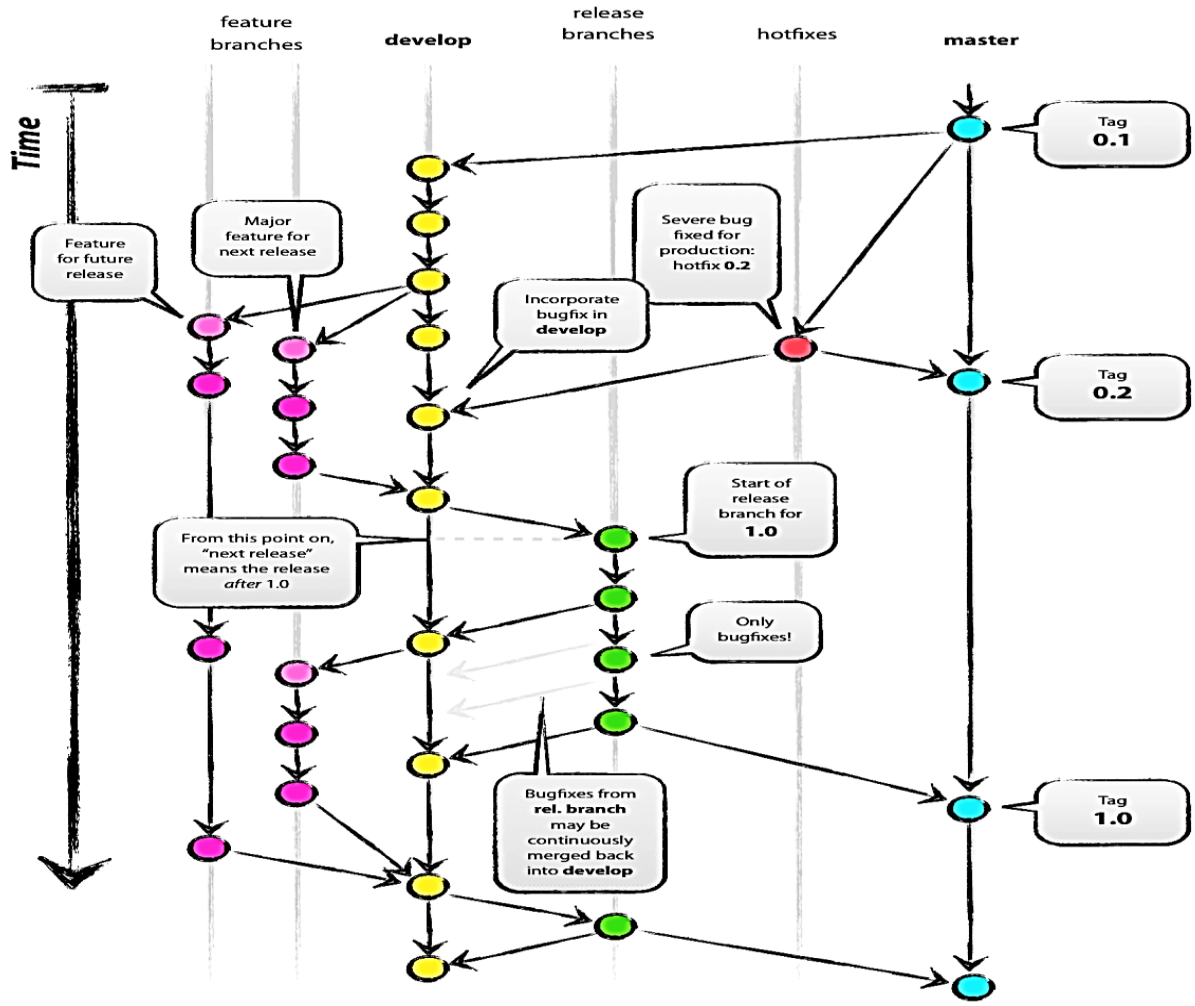


3. Hard (Updating the Working Directory): Here total Head, Index and working directory moves to previous one. looks like the
$ ***git reset --hard HEAD~***



https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified

GIT FLOW



- **Main Branches**
  I. Master branches
  II. Develop branches
- **Supporting Branches**
  I. Release branches
  II. Feature branches
  III. Hotfix branches

Master(Main)---> Develop(Main)---> Release(Multiple Br)---> Feature(Multiple Br)---> Hotfix(Multiple Br)

1. FEATURE BRANCH: Developer pull code from develop branch and work on Feature Branch. Once work is done, developer merge the code with Develop Branch

- May branch off from: **develop**
- Must merge back into: **develop**
- Branch naming convention: **feature**

Create Feature Branch

$ ***git checkout -b myfeature develop*** # Switched to a new branch "myfeature"
$ ***git checkout develop*** #Switched to branch 'develop'
$ ***git merge --no-ff myfeature*** #Updating ea1b82a..05e9557
$ ***git branch -d myfeature*** #Deleted branch myfeature (was 05e9557).
$ ***git push origin develop***
--no-ff flag causes the merge to always create a new commit object

2. RELEASE BRANCH: Release branches support preparation of a new production release. Release branches are created from the develop branch. Automatic testing is done on this branch like (Selenium, PERT…) Once it passes all the test a release tag is given to this and merge back to Develop and Master branch. For Master branch a tag is given with a version and final release is deployed to Staging, UAT, Pre-Prod, Production.

- May branch off from: **develop**
- Must merge back into: **develop and master**
- Branch naming convention: **release-***

$ ***git checkout -b release-1.2 develop #***Switched to a new branch "release-1.2"
$ ***git commit -a -m "Bumped version number to 1.2"***
$ ***git checkout master*** #Switched to branch 'master'
$ ***git merge --no-ff release-1.2*** #Merge made by recursive.
$ ***git tag -a 1.2***
$ git checkout develop #Switched to branch 'develop'
$ git merge --no-ff release-1.2 #Merge made by recursive.
$ git branch -d release-1.2 # Deleted branch release-1.2

3. HOTFIX BRANCH: Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

- May branch off from: **master**
- Must merge back into: **develop and master**
- Branch naming convention: **hotfix-***

$ *git checkout -b hotfix-1.2.1 master* #Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
$ *git commit -a -m "Bumped version number to 1.2.1"*
$ *git commit -m "Fixed severe production problem"*
$ *git checkout master* #Switched to branch 'master'
$ *git merge --no-ff hotfix-1.2.1* #Merge made by recursive.
$ *git tag -a 1.2.1*
$ *git checkout develop* # Switched to branch 'develop'
$ *git merge --no-ff hotfix-1.2.1* #Merge made by recursive.
$ *git branch -d hotfix-1.2.1* # Deleted branch hotfix-1.2.1 (was abbe5d6).

BRANCHING STRATEGY

- Master is a branch where customer releases are done.
- Develop branch where developers work is kept.
- Hot fix branch here customer feedback and issues are fixed.

- Devops Engineer creates Day Builds and Night Builds on Develop Branch by Poll SCM or GIT Hooks.
- Devops Engineer Has to create Release Branch and has to merge develop branch with Release Branch.
- Every release will be a sprint Branch. We configure Night Build on Release or sprint Branches.
- Day build is part where we give feedback to developer whether the code is merged correctly and working correctly or not. In day build we do basic testing.
- Night builds is a part where we give the release code to testing which takes much time. Testing like selenium, pert all automatic testing.

All the developers take code from Develop Branch and create their own feature branch with feature number. They work on feature branch once done merge it to Develop branch. Day builds are done on Develop branch if any build breaks it's a high priority issue. Those are reverted back to developer for fixing. Once all the development is finished, we create Release Branch, based on scrum Master/Project Manager instructions. Release Branch has to be very stable. Any fixes in release we work on it. After everything is ok, we merge it to Develop and Merge Branch with a tag. Customer will get release from master. Any fixes from customer is taken from master and pushed to Hot fix. Fixes there and again merge back to develop and master.

**All the Release, Develop and Master Branches are configured from Jenkins

- Master Branch will always be taken to Staging Environment and Production Environment
- Master Branch we do night builds for two - three weeks with manual trigger. we do deployment to Staging, UAT, Pre-Prod, Production
- ❖ Release branches we create night builds only when there is a change (manual trigger). Deploy to QA environment
- ➢ Develop Branch both Day and Night builds. We automate triggers. we config dev and component testing (done by scrum team tester).
- ✓ Day builds we do MVN package, testing and share test results (project initial days only we create deploy).
- ✓ Night builds we do MVN Package, testing and deployment.

Multiple files added to git. One of the files want to take out of staging area?
If you need to remove a single file from the staging area, use
$ git reset HEAD -- <file>
If you need to remove a whole directory (folder) from the staging area, use
$ git reset HEAD -- <directory Name>

How to remove a file from git
git rm file1.txt
git commit -m "remove file1.txt"
But if you want to remove the file only from the Git repository and not remove it from the filesystem, use:
git rm --cached file1.txt
git commit -m "remove file1.txt"

Git folder name is changed after cloning. One of your team mate asked to share the link. How will i get.
ANS: Go to. git folder and check from where last commit is done.

How to clone a specific branch or master Branch and the last commit only.
git clone --depth=1 --branch dev https://github.com/your/repo.git

What is Sub-Module and why we need sub module
Git submodules allow you to keep a git repository as a subdirectory of another git repository.
Git submodules are simply a reference to another repository at a particular snapshot in time.
When adding a submodule to a repository a new. gitmodules file will be created. The. gitmodules file contains meta data about the mapping between the submodule project's URL and local directory.
git submodule add https://bitbucket.org/j
git clone /url/to/repo/with/submodules
git submodule init --- to copy the mapping from the. gitmodules file into the local ./. git/config file
git submodule update

https://www.atlassian.com/git/tutorials/git-submodule

Command to list all the Branches
$ ***git branch -a*** ---> -a shows all local and remote branches
$ ***git branch -r*** ---> -r shows only remote branches

How to delete local branch and remote branch
Ans: Deleting a branch LOCALLY:
Git will not let you delete the branch you are currently on so you must make sure to check out a branch
$ ***git branch -d <branch>.*** ----> -d option will delete the branch only if it has already been pushed and merged with the remote branch
EX: git branch -d fix/authentication
$ ***git branch -D <branch>.*** -D instead if you want to force the branch to be deleted, even if it hasn't been pushed or merged yet.
Deleting a branch REMOTELY:
$ ***git push <remote name> --delete <branch name>***
OR
$ ***git push <remote name> :<branch name>***
If you get the error as push failed then use
$ ***git fetch -p*** ---> -p flag means "prune". and try to sync the branch list.

// delete branch locally
$ ***git branch -d <localBranchName>***
// delete branch remotely
$ ***git push origin --delete <remoteBranchName>***

Diff b/w git diff and git status
git status only shows the modified files which are to be added
git diff shows the file content and matter that was modified by comparing with previous commit.
NOTE: $ ***git status -s*** and $ ***git diff -name-only*** both commands serves for same purpose

What are hooks in Git?
Git hooks are scripts that run automatically every time a particular event occurs in a Git repository.
Git Hooks we will find in. git folder > hooks
Two kind of Hooks
1. Client Hooks -
2. Server Hooks - is used to call Jenkins by adding Jenkins URL in GitHub hooks path.

CONFLICTS IN GIT:

Conflicts in git occurs while merging the branches. This is because of difference in code of two branch. This can be resolved by editing the conflict file. In conflict file the code has to be placed in sequence in with the merging branch.
Then make a commit and merge the branch. Conflicts will be resolved.
Mostly merge conflicts related to code will be resolved by developers.