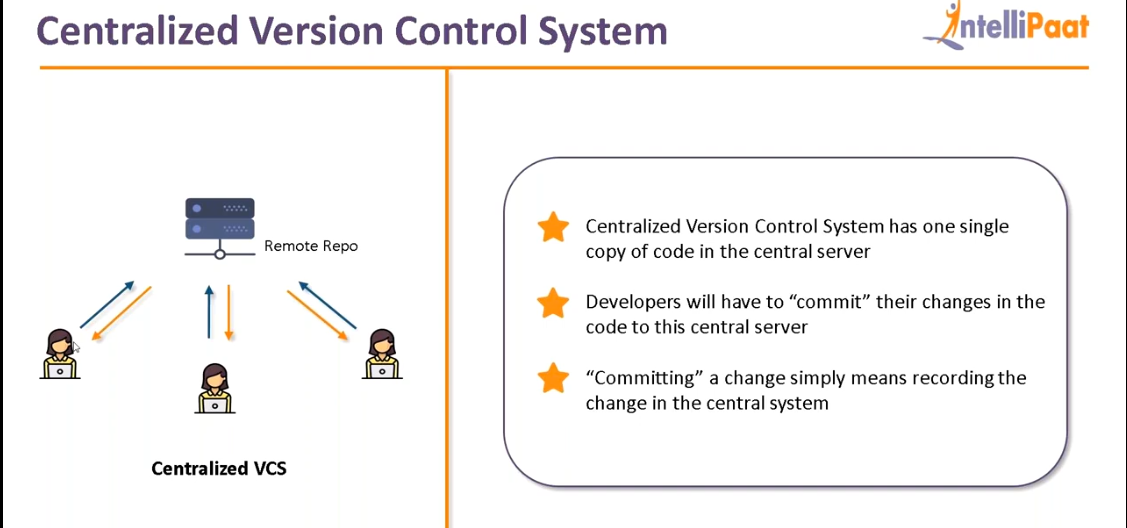
===========================================

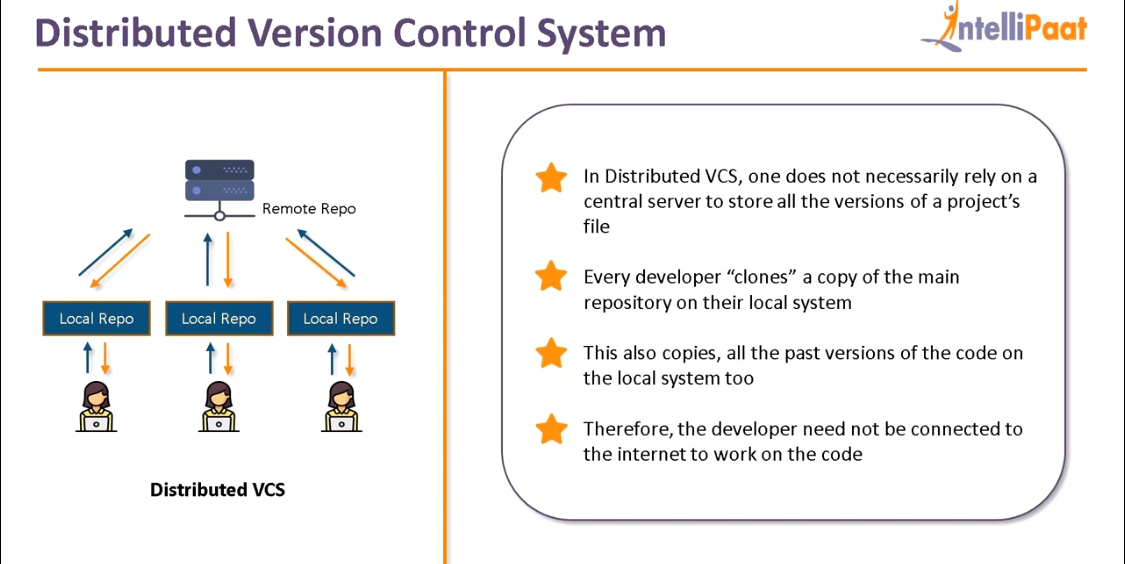
Code repository:

Version Control SYSTEM

CVCS:



Tool for CVCS is SVN



Managing the version

1. Local versioning DB -->Mulitpeople cannot use it

2. Remote repo or Central VCS: Performance issues,anybody can access,

3. Distributed VCS

Local+Central -->Distributed

EX: GITHub,gitlab,bitbucket,tfs,code commit.

More HA, high performance.

====================================

GIT

GitHub:-->remote repo

Local repo-->git client

remote repo(multiple repos)--->Git hub server

^

|

|

|

Local repo(multiple repos)-->git client

Local repo --->Push --->Remote repo

Local<---Pull / clone <---Remote repo

**Fetch**: in order to sync up remote repo and local repo we use fetch.But we cannot see the changes.

**Pull**: fetch + merge to sync remote repo and local repo and also we can see what changes are pulled.

If you create a directory and clone a repository using git clone. Then we don’t need to add the remote origin as it will automatically pickup the parameters.

Difference between clone and pull is clone is used to get the replica of entire repository where as pull is used to update the repository. If local has 4 files and remote has 5 files then using pull we can update the local repo with the latest commit 5th file.

To create local repository use git init in that repo or folder or directory:

**Git init** -->create empty git repo

to store files to local repo we should commit

**git add <filename>** --> to move to staging area from unstagged area or from untracked to tracked files

**git commit -m “first commit”**-->to save files from working directory to git repo (local repo)

**git remote add origin https://github.com/alokvk1234/git1.git** (git hub url where changes can be pushed)(to add remote repo to local repo)

**git push -u origin master** -->pushing the commits from master branch to origin

**git remote -v** -->to check which remote repo it was pushed

cat .git/config

to make passwordless authentication with git hub while pushing we have to generate public and private key.This is done in git local repo. It can be done as below.

**ssh-keygen**

**cat id\_rsa.pub** -->paste the contents in github repo in settings -->SSH and GPG keys.

**git remote remove origin** -->to remove remote repo

**git pull origin master** -->to pull from remote to local repo

**Scenario:**

You have already files in staged area and again you create some files in unstaged area or working directory. Now if you want to ignore these files during git add command then use .gitignore file.

touch .gitignore

vi .gitignore

Add the files which you want to ignore during git add command.

**Scenario:**

If we perform changes in git remote repo and if the changes done in remote repo are not done in local repo and if you try to push from local to remote then it will be rejected. Means there must be sync between local and remote repo before doing any push. We have to fix the conflict.

**git stash** -->used if you want to ignore some un staged content in local repo and git will move them to temporary repo. Later we can pull from remote and stash with local. This will avoid extra commit.

For example if you have created master branch which has 1.txt, 2.txt and 3.txt files. Now you create new file 4.txt and now you want to switch to other branch feature where you have some other task to be completed. Now in feature branch also you find 4.txt as it is not tracked yet. So in order to avoid the confusion we will stash the file 4.txt in master branch to some temporary directory by **git stash** command before moving to feature branch. So that when you move to feature branch you won’t see the 4.txt file which will avoid confusion. Also once your task in feature branch is completed you can switch back to master branch and get back the stashed files using below command **git stash pop**.

**git pull** -->will sync up remote to local and merge the code.

So if you want to avoid merging then stash the content from local repo before add and commit.

**git stash clear**

**git stash --list**

**pull=fetch+merge** (pull from remote to local and merge with local)

**git reset <commit id1>** -->delete complete content with commit id of commit id2 and head will reset to commit id1.

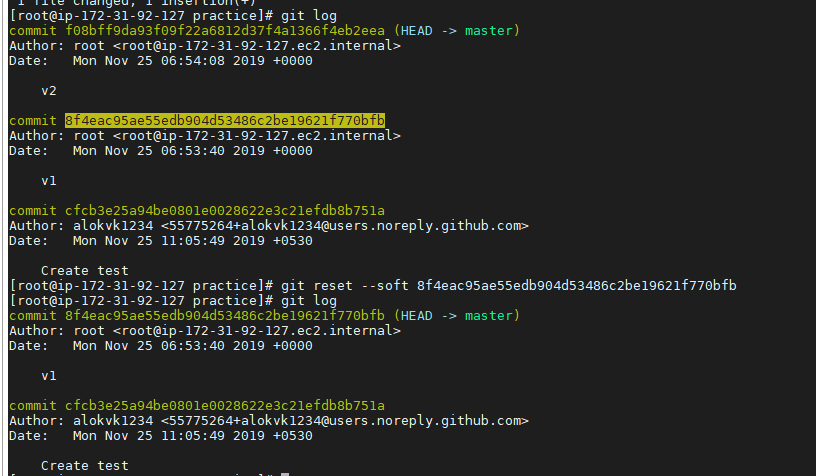
Ex:

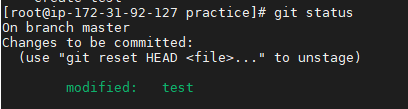
<commitid2>

<commitid1>

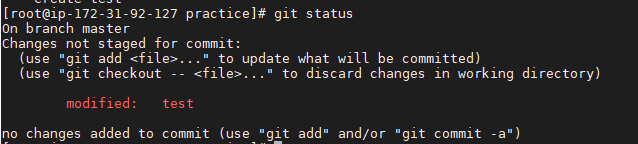
**git reset HEAD <filename>** : If a file is already moved to staged area but not committed can be brought back to unstaged aged using this command.

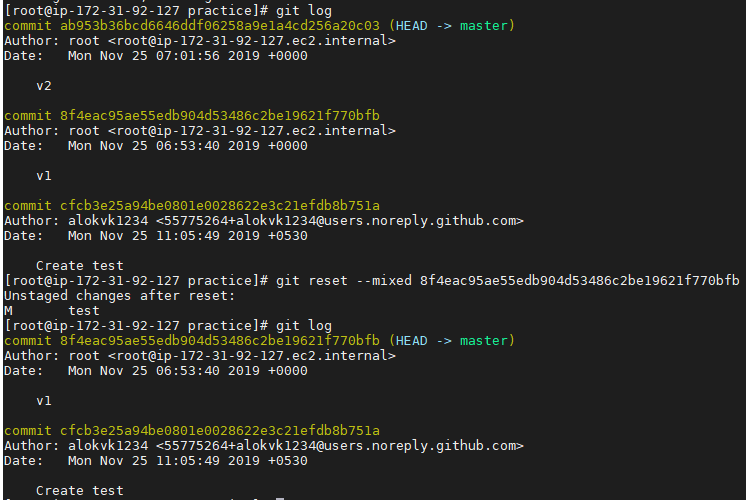
**git reset --soft <commit id>** -->will delete content but will be in stages area (green colour).In git add level but not committed. In below screenshot issued command git reset --soft <commitid of v1>. So HEAD is reset to v1 commit id from v2.Also the changes made in v2 commit will be moved to staged area but not committed (in green colour).





**git reset --mixed <commit id>** -->will delete content but will be in working area (red colour). before git add level not yet git add and git commit.(default). In below screenshot issued command git reset --mixed <commitid of v1>. So HEAD is reset to v1 commit id from v2.Also the changes made in v2 commit will be moved to unstaged area but not committed (in red colour).





**git reset --hard** **<commit id>** -->Commits and files going to delete. Content will neither in red or green instead vanished. It is default.

**git reset ~HEAD** -->Latest commit. Head is pointer to latest commit.

**git push origin +master** (forced update) -->If you have 11 commtits in github and 10 commits in git client. If you issue this command all the 11 commits will be deleted and 10 commits from client will be pushed to github.

**git revert <commit id2>** --> used to temporarily ignore that commit and will point to previous one. Here if you issue **git log --oneline** then file with respect to commit id will be there and another commit will be created with revert. If you view the file in directory you cannot see content related to commit id2.

Ex:

<commitid3> Revert "Create jenkinsfile"

<commitid2> Create jenkinsfile

<commitid1> update pom.xml

Now it will be pointing to <commitid1>

Again if you give **git revert <commitid3>** then it will be revert of revert and we get back the original file (Create jenkinsfile) back but there will be new commit id created.

Ex:

<commitid4> Revert "Revert "Create jenkinsfile""

<commitid3> Revert "Create jenkinsfile"

<commitid2> Create jenkinsfile

<commitid1> update pom.xml

**Branching strategy:**

Default branch is master.

**git branch** -->will show existing branches

**git branch branch1** -->will create new branch branch1

**git branch -D branch1 🡪**will delete branch branch1

**git checkout branch1** -->then the asterisk will be pointed to branch branch1 or switched to branch1.

**git checkout -b dev** -->new branch dev will be created and asterisk will be pointed to dev branch or switched to dev branch.

During branching the last commit of master branch will become first commit of feature branch.

git log will give log of what branch are you on. Like if you’re in mater branch you will get master branch logs.

Note:

**Git branch branch1**

**Git checkout branch1** here new branch branch1 will be created and switch will be at branch1 (asterisk will be at branch1).

Now again if you create a branch2.

Note: If you want to checkout to different commits then use below command **git checkout <commitid>**

**git branch branch2** then branch2 branch is created from branch1 branch and all the contents from branch1 will be copied to branch2 and not master.

Note: Head -->Latest commit.

To merge the code from once branch to another we have 3 ways:

1. Merge

2. Rebase

3. Cherry-pick

Merge: First checkout to dev.

git checkout dev

so now you’re in dev branch.

git merge feature1 -->feature1 is source branch. Now feature1 will be merged to dev.

In real time we cannot directly merge the feature branched to dev branches because of branch protection rules set in git hub. This can be done by pull requests in git hub.

Merge -->extra commit will be created --> we will come to know from which branch

Rebase -->no extra commit will be created -->clean history or linear history

We don’t have permission to rebase or merge from downstream to upstream. But can be achieved using pull request from github.

We can make cutover on commits on a particular branch by creating tag. Cutover can be done by team leads.

**git tag <alias name> <commit-id>**

ex: git tag prod-1.0.0 b1ce2d7

Later we can push to master (github) using below code

**git push origin --tags**

**Fork:** If you want to avoid too many branches we can go for forking. It is just like clone. If you want to have a copy from server side then you can go for fork.

It is like duplicate of your repository, work on it and issue pull request to merge to main repository. You will be having admin permissions.

**git mergetool** is to represent UI for merge conflicts.

**git cherry-pick <commit-id>**-->similar to merge and rebase. If you want to merge a particular commit in another branch. Here if you have switched to master and if you issue cherry-pick for commit-id then only that particular commit-id will be merged with master.

**git diff commitA commitB** -->will display the difference between commitA and commitB.

**git diff HEAD .** 🡪Will display the difference between latest commit and the current working directory

git diff HEAD~2 🡪Will compare latest commit with latest -2 commits.

**Hot fix:** If you have a release rel1.0.0 going to production and if it is released. Later you identify bugs in the release so we will take a hotfix branch and fix it and release it as patch release.

GIT is most popular tool among all DVCS tools.

Used for source code management.

Creating repositories

Making changes

Syncing repo

Parallel development