Git is a local VCS.

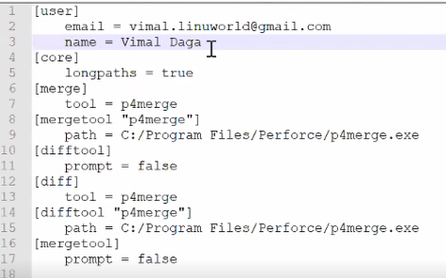
GitHub is a CVCS

**Day 1**

* Suppose you have created a website and put in your production environment.
  + After sometimes your developer has a created a new feature in the website called login feature.
  + Because of some technical error your app goes down after updating.
  + So we can do one thing.
  + Go back to the last running website state.
    - For this we can do undo but if we have bigger code, undo will not work properly.
    - Before adding new feature create a backup for last data.
* In git world these areas where you create backups known as commit area.
* Git is a source code management tool.
  + It provides facility for VCS (version controlling system).
  + Rollback or rollout to new version.
* Developer has to decide when to create backup.
* You can save image & video also.
* Between working area and committing area we have one more area.
  + Now onwards git will keep on tracking files.
  + When you write delete or modify something they will track per line basis.
  + Middle area (add) contains one database, they don’t save your working file or backup file.
  + Database contains change you made to the file.
  + This is stored in staging area.
  + For this we have to tell git to track this file.
    - For this we have to use add keyword.
  + So, It is not known as backup it is known as point-in-time backup or snapshot.
* Suppose you have one backup and your developer wants one more backup.
  + Rather than creating backup and wasting our storage, only update those information which is changed or new created codes.
  + Only updated part will added to the second backup.
    - But when you see it will give you this data and also reference to first update.
    - You can see whole second backup.
    - This thing is known as point-in-time backup.
  + One tracker is running behind the seen for this.

Practical part

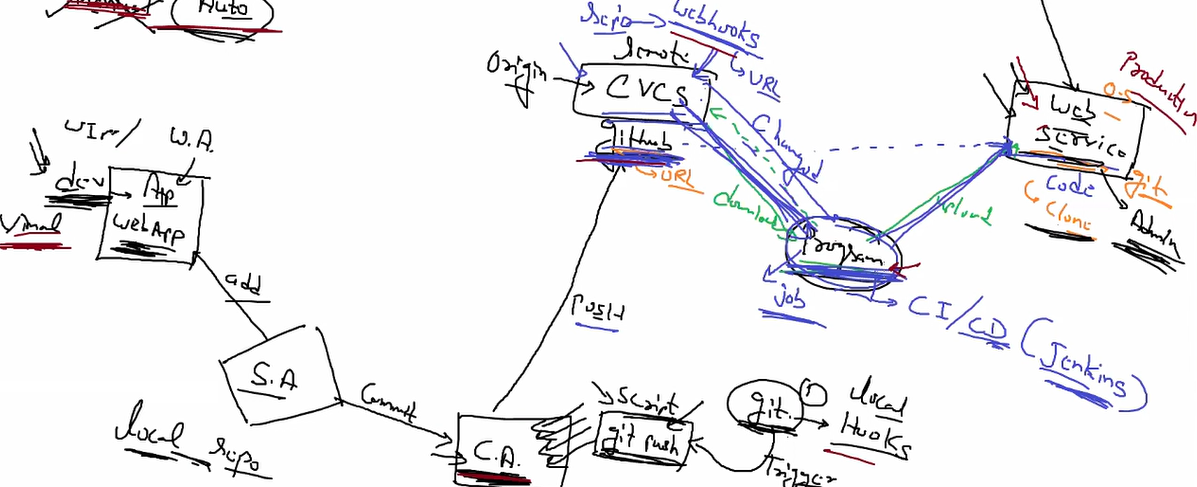
* Git only maintain commit area and staging area.
  + For this first you have to init as a git repository.
* When you commit they will assign a random number, for uniquely identify that version (version number,commit Id).
  + While commiting it is compulsory to give some comment
  + They also give you commit date & time.
* Git init
  + Create a new empty repository for git.
  + You can check this using ls -a
* Git add file.py
  + Now they are started tracking this file.
* Git commit main.py
  + It will open your default text editor and here you have to type your comment.
  + You can also comment in terminal.
  + Git commit –m “sec commit” main.py
* Git status
  + It will show you if you have something modified in staging area and not commited.
* Git log
  + It will show you all commit with time and commit ID.
  + Git reflog
    - Give you logs in short.
  + Git log --online
    - Commit ID & comment.
* Currently head is pointing to the latest commit.
  + This is one keyword or pointer.
* Git reset <first version commit ID>
  + It will reset your pointer to first version.
  + You can write 4 characters instead of full commit ID.
  + Reset is for rollback.
* Git checkout -- main.py
  + Now head will point to that version where we have reset.
* Here we have to provide entire code access to our developer.
  + So instead of giving main code we can give exact copy of the code, this concept is known as branch.
    - Also known as feature branch because we created this to add some features.
  + We can achieve one more thing here.
    - Now when you create new feature instead of direct updating your main production, we can check it in different branch and OS.
    - If it will work fine we can merge it to the main branch.
    - After merging we can delete this branch or keep on working on this branch.
* By default when you start working with git init, they will automatically create one branch known as **master branch**.
  + It is main branch.
  + Master branch data we don’t change directly.
  + Master branch have all the commits.
* Feature branch is most of time ahead from master branch.
  + So they are known as upstream branch/project.
    - Master branch = 3 commit
    - Feature branch = 3+2 commit
      * 2 from developer new commit after branch.
  + When we merge this to master branch, master branch also have 5 commits.
    - Most of time we use master branch for production, but it is not compulsory.
  + Every branch has their own personal timeline.
* Git branch
  + It will show you all branch.
  + Git branch -a
    - It will tell you all branch you have.
* Git checkout –b dev1
  + Create a new branch and go to that branch.
  + You can check with git status.
  + Entire history is copied to dev1 branch.
* Git log
  + You can see all commits.
  + It will also show you master and dev both branch heading towards same commit.
  + Add something and commit.
  + Now when you see git log, master branch & dev branch pointing to different version.
    - Dev branch is ahead from dev branch.
    - Git log --online
* Git checkout master
  + Now you cannot see new data available in dev branch.
  + Git reflog, now you can see something different between log and reflog.
* Git merge dev1
  + It will merge dev1 branch to master branch.
  + Dev1 branch history is updated in master branch.
  + Now master branch can also see all data of dev1 branch till last version before merge.
  + This type of merging strategy is called Fast-forwarding.
* We can add tracker to dev branch so master branch can know what developer is doing.
  + This thing is known as upstream tracker.
  + Git branch --set-upstream-to=dev1
  + Git branch -u=dev1
    - Both are same commands.
    - Master branch add a tracker for dev1 branch.
    - What if we run this in dev 1 branch add tracker to master branch?
* Git status
  + Now they will say you, you (master) are 1 commit behind.
* Git pull
  + It will only work when your current branch knows what is your upstream branch.
    - For this we have to add tracker for upstream branch.
  + They will also merge with master branch.
  + They will use fast-forwarded strategy.
    - Fast-forward means they will merge all the commit append to particular branch where you are.
* Here we have created local VCS, maintained inside .git folder.
  + If you want to share this repository with someone or to public world.
  + For this we have lots of cloud provider, one of the famous one is **GitHub**.
* If you want to create Git repository in GitHub.
  + Create a new repository.
  + Give unique repository name.
    - It is something like creating a workspace.
  + Click on add a README file.
    - It will create an empty git repository.
* We want to upload local git repository in remote location-GitHub.
  + Don’t click on README file, we only need ws, we don’t need repository in GitHub.
* Git remote -v
  + But we don’t specify remote location, so we have to specify it.
  + Git remote add mygithub https://github.com/akshit0704/demo.git
    - Now we have added this ws in this file, local repository.
    - We have to give name to every remote repo.
    - Uploading something to remote ws known as **push**.
    - Now check with git remote -v
  + Git push mygithub master
    - You have to mention which branch you want to push and which remote location you want to push.
    - When you push internal .git folder also uploaded.
    - Do second commit and check logs with git log.
      * Only first commit heading to remote location, second is not pushed to GitHub.
    - Add something in file using GitHub.
    - Now GitHub has new data, but our local repo don’t have this data, we have to pull this from Github.
  + Git pull mygithub master
    - Sometimes someone has changed In GitHub and you have changed in the local.
    - Now if you use git pull it will conflict.
    - So instead of pull complete data only pull commit history, if we have all things fine then pull complete data.
  + Git fetch
    - But it will not work, you have to tell what is your upstream branch.
  + git branch --set-upstream-to=mygithub/master master
    - master in github is upstream branch.
    - Master is local branch.
  + Git fetch
    - It will give you new commit is made or not.
    - If all things are fine you can merge this using git pull.
  + Git show <comit\_id>
    - This will show you what you have done at that commit point in time.
  + Git diff <comit\_id><comit\_id>
    - It will show you difference between thesetwo commits.
  + Create a new branch dev1 add some code & commit.
  + Git push -u mygithub dev1
    - It will add tracker.
    - Create branch in Github dev1 & push there.
    - Git push -u mygithub dev1 from master branch will upload master->dev1?
  + Git branch -a
    - It will show you all branches.
* Merge dev1 branch to master branch using GitHub.
* Go where you want to merge.
  + We want to merge dev1 in master.
  + Go to master branch.
  + Click on compare & pull
    - They will automatically detect which merge strategy use.
  + Click on create pull request.
  + Click on merge pull request.
    - You can also delete feature branch after merge.
  + Git config --global –e
    - Here you can create global file, which will work for all repository.
    - It will pop up a file.
    - You can use local keyword for per repository file.



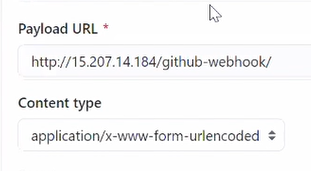
* + - Now in commit you can set your auther name.
  + Check with git log.
* Download repository from Github.

**Day 2**

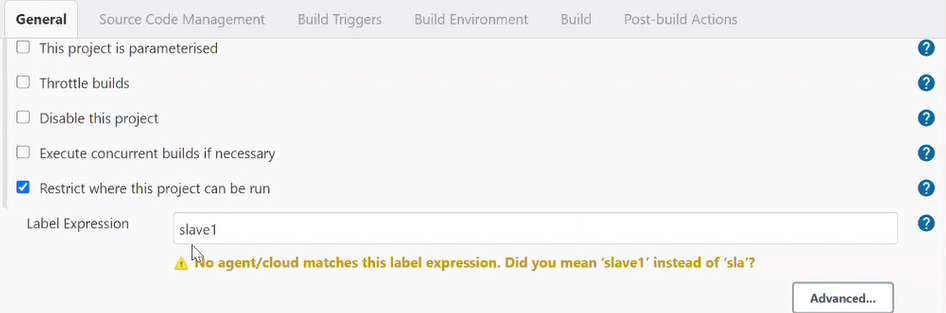
* Version 2 of same file has only **delta part** of version 1.
* Normally we will use master branch for production, so it is also known as release branch or base branch.
* Git cat-file -p <commit\_id>
  + To see metadata of a timeline.
* When you initialize git, it will create a tree (give a name/id to tree) and on top of tree git will create versions.
  + Git add ….
  + Git commit …
  + Git cat-file -p <commit\_id>
    - This time you can see parent of that version.
* Sometimes we want to remove new data, only want older data which is already commited.
  + May be because of some issue or mesh we want to remove new things.
  + Git checkout a.txt
* We use local git, it will keep on track commit area and when someone commit they will automatically push to GitHub (other CVCS).
  + They will trigger some script for us.
  + We have to set local **hooks** in the git.
  + And code will updated to GitHUb.
  + GitHub **triggers** some CI/CD tool (Eg, Jenkins).
  + It will go to Jenkins and tell him something changed.
  + Jenkins comes, get data and updated new version app available in production system.



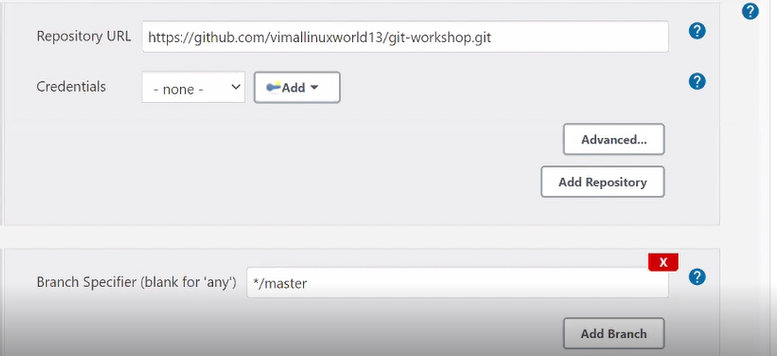
* **Release branch == master branch == base branch.**
* Clone from github.
  + Git clone <url>
  + Change something
    - Git add …
    - Git commit ..
  + It is launched from github so it will automatically add upstream branch.
* Practical
  + Create one demo page in GitHub.
  + Use Git Clone for cloning inside your local Git.
  + Go to github -> Settings -> WebHooks.
  + Type Jenkins API in payload URL.

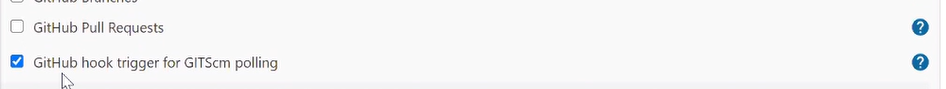


* + Create a job in Jenkins.
    - Add git as SCM & give your repo URL.



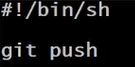
* + - Slave is our end webserver.



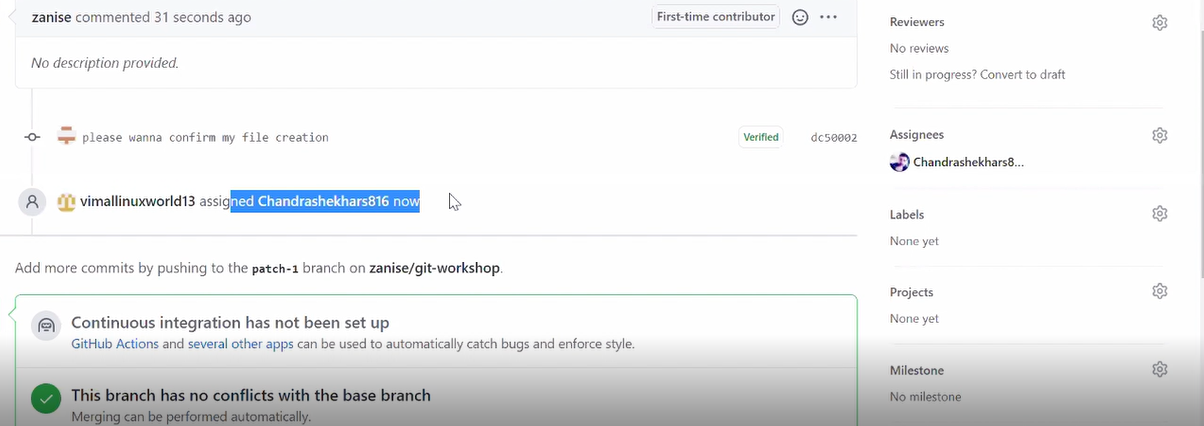




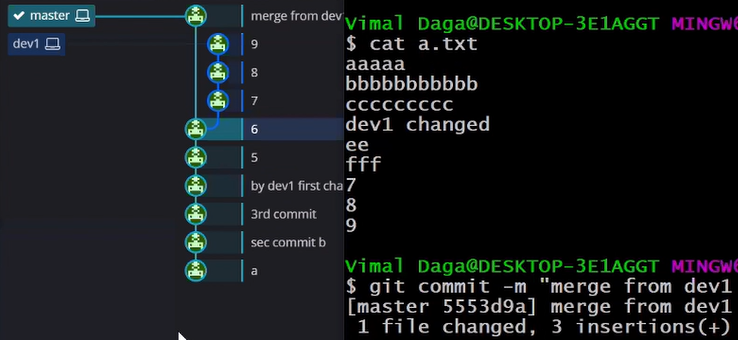
* + Use hooks inside local git.
    - Cd .git/hooks
    - Here we have to create a file for post-commit.
    - Write below thing and store as post-commit



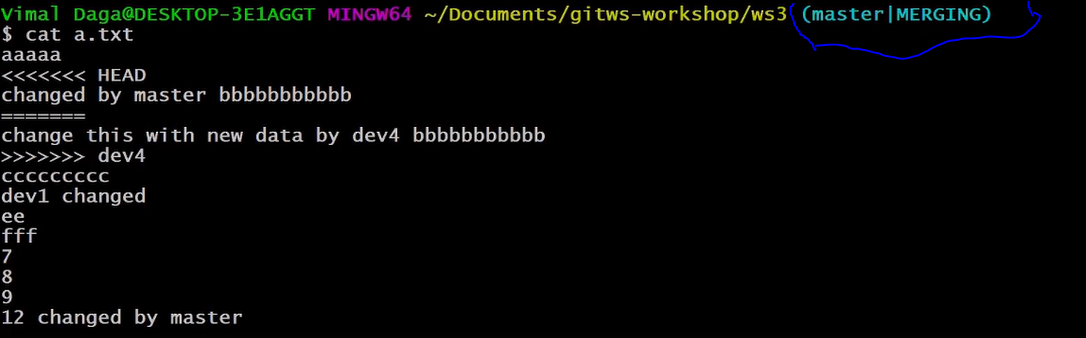
* + Now when you commit it will automatically pushed to github using hooks and GitHub will trigger Jenkins and Jenkins will update our code.
* How can we contribute in GitHub?
  + From your account go to the GitHub URL you want to contribute, **fork** (copy) it in your repository.
  + It will create a copy in your account.
  + It is like your local repository, you can do all the things here.
  + You can also add some code and commit in your local (forked) repository.
  + Create some versions.
  + Click on open a pull request, create a pull request.
    - GitHub will auto detect conflict occur or not.
    - Now main account developer has to decide accept this request or not.
  + Developer will click on merge pull request.
* Developer can also create a team who can manage merge request accept or not.
  + For this developer has to go to manage access and add username or email ID.
  + After this go to merge request and assign to this person.



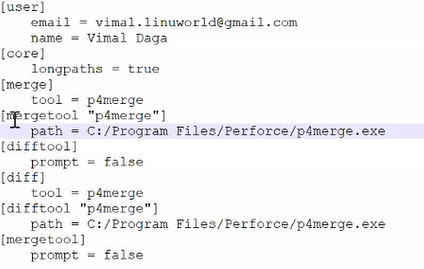
* Git kraken is a graphical tool for git.
* Git switch master.
  + Change branch.
* Suppose from master branch we have three commits, then we create dev1 branch, he has created 3 new commit and merge with master.
  + Here master will use fast forwarding and has complete timeline of all 6 commits.
  + But sometimes we don’t want to get all commits.
    - We want only last commit (only data) of dev 1 branch, total 4 (3+1) commits.
    - More commits sometimes decrease performance.
  + If you don’t want to get complete timeline, in this case you have to use squash.
  + Git merge --squash dev1
    - It will create a copy of last version from dev1 in working area.
    - So if all things are fine master can commit.



* + - Here you can see master get 3 versions data in their w.a. and he commit with a new version.
* Suppose you have 4 commits in master, then you create dev1 branch and 3 more commits.
  + But master branch also has one commit after 4 commits.
  + Here master branch is known as base branch.
  + For dev1 branch base branch is master with 4 commit but master has do one more commit.
  + So you have to change base to 5th commit of master.
  + Because you want to get all data from master.
  + Git rebase master
  + We normally do rebase in feature branch.
* Suppose master has 3 commits m1, m2, m3 and then we have dev1 branch commits d1, d2 and master has commit m4.
  + So this time when you merge it will use **recursive strategy** and known as **3-way merging**.
  + Here after m3 they add d1 & d2 then m3.
  + Here they have created one extra commit known as merge commit at end.
    - So we can use this for rollback or something change.
    - Here merge commit has 2 parents m4 & d2.
* Suppose developer has create 3 versions, z1, z2 & z3 in dev1 branch.
  + But master wants to get data only z1 commit, because it have some good features, other things are still going on.
  + So he want to get data from z1 in master branch.
    - This is known as **cherry pick**.
    - Git cherry-pick <z1\_commit\_id>
      * It will pick version z1 copy data in master branch, and master will create a new version of it.
* Suppose master & developer both has different timeline.
  + Both have changed in same file & same line.
  + Now they have commited & they want to merge it.
  + Here when you commit it will show you some **conflict** error.
  + Now you can see you are in some other **master-merge** branch.



* + You cannot do anything here.
    - First you have to solve conflicts.
    - You can manually check file & remove from master or developer.
  + After changing you have to write
    - git add .
    - git commit
  + here you don’t have to provide any commit name.
* we also have graphical tools for solving merge conflicts.
* Git mergetool
  + It will launch tool (p4merge in our case).
  + Change and save it.
  + After this you have to commit only.
    - Sometimes we require old data, so you can use backup it has extension called .orig
  + For set p4merge as merge tool you have to use config file.
  + Git config --global -e



* + We can also use p4merge to differentiate between 2 versions.
* **Stash** 
  + Suppose you have created multiple files, you have commited it, again change in some fie and now you want to merge it.
  + But you want to merge only some files, you don’t want merge all files, because some files are not completely workable they are still in progress.
  + When you commit & merge it will merge all things to master branch.
  + Stash is like one area, where you ask your working area to memorize the content you have changed but not commited.
    - * It will not commit.
    - Git stash save
    - Git stash list
  + Now if you see cat file.txt
    - It will not show you code write after commit.
    - It is saved in stash and you can do whatever you want to do.
  + Merge with master branch.
    - All necessary things are merged, but those files are not merged.
  + For getting data back you have to use git stash apply.
* Suppose you have created 4 commits in master.
  + Now your head points to latest commit.
  + But you want to remove 2 commits.
    - Git reset --mixed HEAD~2
  + By default reset type is mix.
    - It will remove data from commit area & staging area but it is available in working area.
  + We can use this in multiple situation.
  + Suppose you want to create new branch but don’t want to give him last commit data.
    - So roll back using reset create branch & again commit, this time master has all data commited, but dev branch don’t have this data.
  + Mixed – remove from CA & SA
  + Soft – remove from only CA
  + Hard – remove from CA, SA & WA
* Some version you have in local system but in GitHub you have different version because of reset.
  + So here also merge conflicts come at push time.
  + So instead of reset you can use **revert**.
    - **Git revert <comit\_id>**
    - They will add extra marker on top of this commit id that this file deleted.
  + Now if you push it will not give you conflict error.
* In other VCS tool we require git centralized server server.
  + Developer 1 can create their own server & dev 2 also create their own server.
  + Now instead of going to centralized server dev 2 can directly connect to dev 1 server and pull data.
  + Peer to peer connectivity.
  + Because of this git is also known as distributed
  + When you download git it automatically creates a git server.
  + Git clone [root@192.168.0.129:/wsgit](mailto:root@192.168.0.129:/wsgit)
  + By default they use SSH protocol.
* Git is VCS tool.
  + In SCM we have lots of things, one of the things are version controlling.
  + Sowe can say it will give you scm-vcs but core of git is VCS.
* GitHub is meant for sharing complete VCS timeline to others.