Git

Requirement

- You need Git at your PC
- You need GitHub on the server. Make a GitHub account.
- Once you make a GitHub account, create a repository, under your username.
- On your PC, preferably use GitBash shell and Notepad++.
- Make a clone of repo on GitHub to your PC using personal access tokens.
- personal access token is some token generated with an expiry date (preferably) after passing some string (like blockchain Ethereum).
- Git has stopped supporting username and password since Aug 2021.

Procedure to setup

- create a repo in github
- clone this repo in git using command
- git clone <copy the url address from github itself>
- A folder with the name of remote repo will be created in your git home path. c:/user/pandeype
- create a file and write something in it.
- check git status //it will show the file is unstaged
- git add . and then git commit –m “first file added”
- git push //you will see git prompts for password and username, actually this has been deprecated and won’t work
- So, I will generate Personal Access Token (PAT)
- git push <repo address url> // you will tweek your PAT before git.com in URL.
- https://<PAT>@github.com/PratPro/project_pandeyprat.git
- Issue is each time it will ask for username and password while push, so what you can do:
- `git remote set-url origin https://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@github.com/PratPro/project_pandeyprat.git`
- `If you haven’t cloned then from git:`
- `git remote add origin https://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@github.com/prateekpandeyscholar/ProjectFirki.git`

- git config - - list //shows the configuration file, check your username and email here.
- git config - - global user.name “Prateek” //set username
- git config - - global user.email scholar.prateekpandey@gmail.com
- git config - - global –unset user.email //remove some entry from config file
- This will be the user who will do work and commit on the remote repo.
- User on remote repo can be same or different.

Unstaged, Staged, Committed, and Pushed

- 4 life stages of a file.
- When you create a file in your local repo, it is unstaged by default.
- You write command git add filename.extension→ it is staged
- You write command git commit –m “This is the change I’ve made”
- //above command commits a file.
- git push //push the file from local repo to the remote repo. So consistency is maintained.
- Other commands-> mkdir, cd, ls, git mv, git log, git log –oneline
- //git log gives details of past commits, oneline gives briefly
- //head is the pointer points to the current commit.

Making changes in a file

- You change a file and it is back to the unstaged state. Add it, commit it and push it again.
- Same goes with renaming and deleting a file.
- “mv oldfilename.ext newfilename.ext” combines deleting old file and creating new file in unstage stage. This is renaming.

Travelling in past

- use git log –oneline to get the id of the past you want to travel to
- use git checkout id // to get to the past
- Don’t make changes in the past or timeline will be messed
- instead branch-off from the past using git switch -c <branch-name>
- Make some changes in the past file. add, commit and then push to the remote. If you don’t make any change in the past file, Message will be thrown “Not anything to merge”.
- use git checkout master (main maybe) to come back to the future.
- Best Practices:
- Do commit often
- Commit must be atomic
- Use comments with commit

Tags

- Tags are a label that are attached to a commit to make it easily findable.
- Tags have a name and description (both optional but highly recommended)
- Tags need to be pushed to the remote repo
- Tags are generally created for every new release of your file.
- So Tags are commits having no ordinary value.
- With tags you won’t have to find or type commit id’s to go back to past. Just use tagNames.

Commands with tag

- go back to history using git log – oneline and note any past id….
- use git checkout pastid
- now create a tag here….
- git tag - a tagname - m “description” //when single character for options //you use - . When more than one character you use - -
- go back to future
- git checkout master(or main)
- now I want to go back to past where I also have a tag attached. There are two ways: using checkout and using tag.
- git checkout tags/tagname //and you are there
- you push your tags to remote also
- git push tagname
- git tag //lists all tags
- On gihub you can also check how many tags are there

Main tasks of a version control

- You can go back in time
- You can collaborate

PUll

- git pull fetches the files from remote to local. 3 scenarios exist:
- If file has not changed on the remote since last pull
  - in this case the local file will remain as it is
- If file is changed at the remote but local has not changed since last pull
  - in this case local file is replaced with remote file
- if file at remote and local both changed since last pull
  - in this case merge will take place
- git pull is used for collaboration. git clone is used for cloning repo. you use clone when you start working on a project. you use git pull when you are into a running project. What will happen if you git clone each time instead of git pull? Your current file you are working on will be removed or not ocunted.

Pull merges two files line by line

- You should not make changes to the remote file using GitHub. But you can do it to show demo.
- You cannot git pull until you git commit, otherwise abort error will come.
- If you make changes in remote file and local file, such that the merging is non-conflicting, git is smart to perform merging.
- If merging is conflicting, git opens your files in editor, and ask you to merge them manually, and delete every other characters like <<< >>> that are not supposed to be there in the merged version. You then git add. and git commit and do git pull again.

Binary files(audio, video, images)

- Git merges text files only
- Binary files cannot be merged because they have no lines. And git merges line by line.
- If two files are to be merged you choose one version out of two.
- If tow files are in conflict and you want to merge, may be you would do it manually by opening binary files for example jpeg in phot editor and make changes manually. Then you git add . and git commit

Fetch

- git fetch + git merge = git pull
- fetch does download all the remote files into your local repo, but it does not merge them.
- to do merging, you need to call git merge.
- git pull does the both operations for you.
- fetch and merge gives finer control to us.
- As a programmer, we often will use pull.

Branch

- master(or main) is the default and principal branch that is created when you clone a repository.
- Unless the project is trivial, you never work on master branch.
- You always create a branch emanating from the master, and work on it. Once you are satisfied and committed your work on the second branch, you merge it with the master. Merging with the master is considered a commit.
- You can create as many branches from the master at different points of time, and at sometime you merge them all.

Why do we branch?

- Collaboration is the main reason.
- Consider that we, two people are working on a single project having two modules A and B.
- I will branch-out and work on A; however, my friend will branch-out and work on B. After sometime, I will merge my branch into the master. Then, my friend will merge his branch into the master.
- In software development, two teams seldom work on the same class or file. So merging is often not conflicting.

Commands

- git branch branchName //creates a new branch, but you will still be on the master branch (or old branch)
- git checkout –b branchName //is the mostly used command
- git branch //displays all the branches in your repo, and highlight the current.
- git branch –d branchName //deletes a branch from the local repo
- git push origin –delete branchName //removes the branch from github
- Note: by deleting a branch locally and then git push won't delete it remotely.
- git checkout branchName //moves you to the branchName
- You can only move to another branch, if all the files are committed.
- git push //uploads everything on the server including branches.
- You can delete a branch from the github but the changes will not be reflected in the local repo until you delete it locally.

stashing

- See, you can’t move to a different branch until you commit your current work.
- What if somebody requesting me to switch to his branch and look at what he has done?
- The answer is you stash your current state.
- When you stash, your state moved to the last commit.
- When you come back, pop the stash. And your work after the last commit, will be back in unstaged state.
- Stashes can be many, and stored in the form of stack. So on popping, you always will get the recent most stash.
- You can also name the stash while saving.

commands

- git stash //will create a new stash, and take you to the last commit
- git stash list //will list most recent stashes
- git stash pop // will pop the last stash and restore your files after the last commit in the unstaged state.
- git stash save “stashName” //will save the stash with a name

Merging Branches

- Merging can be done at git and at github as well. Doing so at github is relatively more intuitive and easier.
- When you do a branch merge at git hub, you create a pull request at the git hub (not at local repo or git). Pull request is a way of requesting repo owner to merge the changes of your branch.
- Often someone else will review your changes and approve the pull requests.
- If possible, github will resolve conflicts automatically
- Otherwise, it will let an approver remove conflicts before merging.

Merge vs Sqash

- When you merge:
- you either merge a pull request
- or merge and squash a pull request.

- With merge a pull request, all the commits in the secondary branch will become the commits of the primary branch.
- With merge and squash, all the commits in the secondary branch becomes one commit and then registered as a single commit in the primary branch.
- You often delete the secondary branch after merging.
- Squash simplifies the history.

Merge locally

- you create another branch:
- git checkout –b anotherBranch
- make changes in the file and do git add . and git commit –m “changed”
- You go back to the master: git checkout master
- git diff master…anotherBranch //displays the difference b/w files
- you merge in the master another branch
- git merge anotherBranch //you are already in the master.

Rebasing

- Rebasing is merge from a branch and simplifying history.

- Suppose, you branch-off from main and do some commits on your own branch.
- In the meantime, commits on main also happened.
- Although when I merge my branch to main, things will get resolved (automatically or manually). But in this case history would be a bit tangled.
- If I rebase from my branch using, git rebase main, you would see the commits from main now incorporated in your branch. Later you will push your branch to the github and merge with main.
- Origin signifies your remote repository. You may need to set it at the start. See first-second slides.

When nothing works

- There is no undo button in Git. All you can do is go back to last commit.
- Copy contents of your current repo and save the same in another folder at other place.
  - Clone the repo again for the git hub.
  - Manually merge things from old repo to new repo.
- If you know that git hub has everything latest then:
  - git fetch origin
  - git reset - - hard origin/branchName
    - where branchName is the name of the remote branch
- If you've made changes that you haven't committed yet that you don't like: git reset --hard or git reset - - hard <id of commit that you want to go>

Ignoring Files

- .gitignore file to be added in you local repo.
- It contains entry of files and folders that are to be ignored for tracking.
- add the file to the repository and commit.
- # is a commenting sign in .gitignore file
- temp/ # will ignore all files inside temp folder
- temp/*.txt # will ignore all txt files inside temp
- temp/notes.txt #will ignore notes.txt inside temp
- Example: all build files, node module files
- Software generates build files from source files. Git needs track of source files only. For example: markdown files may be source files for documentation and html files will be generated as build files.
- Files start for “.” are hidden.
- Files that are already committed would not be ignored.

Forking

- Forking is like branching. But branching is a git phenomenon. Forking is a Github Phenomenon.
- You fork a full repository to your own account. Make changes and finally create a pull request to the owner of the repository for accepting your changes. He may accept or reject.
- You generally fork open source projects.

reset and revert

- reset takes you back in history to a desired commit.
- all the commits after the desired commit will be lost.
- reset comes in 3 flavors:
- --soft →your source files will not be lost and will be staged. [Working tree and index remains untouched, head is changed back to last commit. To come back to the starting state (not the desired commit state), all you need is a git commit]
- --mixed (default)→source files will not be lost and will be unstaged. [Working tree is maintained but index lost, head is changed back to last commit. To come back to the starting state (not the desired commit), all you need is git add . and git commit]
- --hard→source files will be lost (updated to the desired commit) [Neither working tree remains nor index, head is changed back to last commit]

revert

- Reset removes commits from the history. Revert doesn’t remove commits, instead create a new commit with older data.
- So, if you have already published you local work to github, and then you figure out that you didn’t want to publish it:
- if you reset in your local repo to the last commit and you push this to the remote, it will throw error. Because the history at github have more commits than your local repo-insconsistency.
- Revert will be fine in this case.
- Reset is okay if you don’t have already published you work, and want to undo it locally.

Practical Use Case for git reset --soft

- Suppose you have a file: A: I am a man. I love Sky.
- Now you think of improving you code and commit it again with tag B.
- B: I am a man. I love Sky. Sky is Blue.
- Now you want to improve it again further with tag C.
- C: I am a man. I love Sky. Sky is Blue. Blue is color.
- Now, you think what the crap is this. I could have easily modified my file single time. I just have one meaningless commit. So I will use reset, remove all the commits after A. My files will be intact and in staged state. I'll just have to commit with tag B. Now I'll have two commits and the desired code.

Course Objective

- To know the necessary about Git
- Without inundating yourselves with the information.
- so that you can start using git and GitHub in your projects.

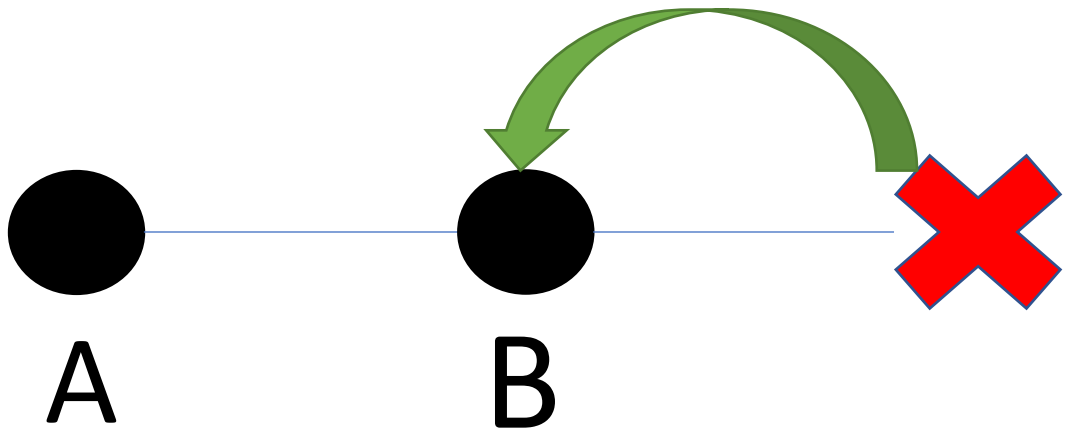
If your answer is yes to any of these questions. You need Git!

- Did you ever loose your code while working on a project?
- Did you ever mess your code so badly that you wished you hadn’t written that messy code?
- Do you want to work as a team on a software project, but you do not know how?
- Do you want to keep all of your projects safe at one place and accessible at all locations ?
- Do you want to show-off your projects and contribute to open-source projects?

What is git?

- Version Control System (VCS)
- Software evolve through versions (V1….V10).
- VCS monitors and maintains the progress of your software during development and after completion of development (i.e. during maintenance phase of software development)
- git is an efficient and popular VCS, which is available to use for free.
- git is often used along with its companion GitHub.
- There are two major objectives of any VCS:
  - Going Back In Time
  - Collaborative Development

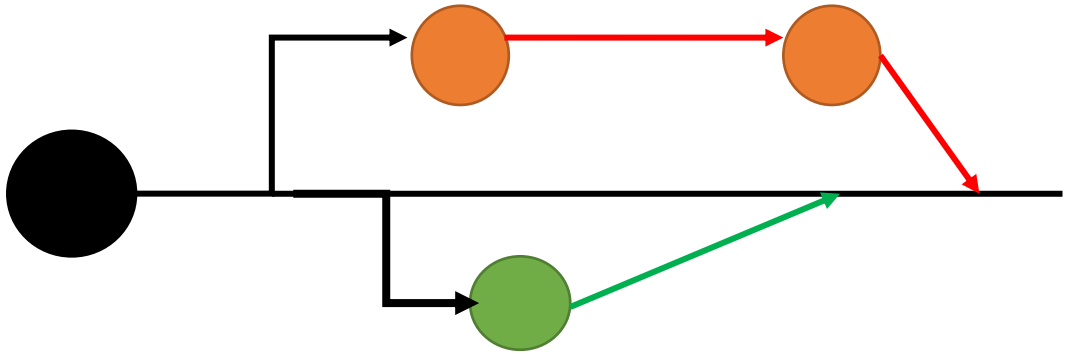
Going Back In Time



git

GitHub

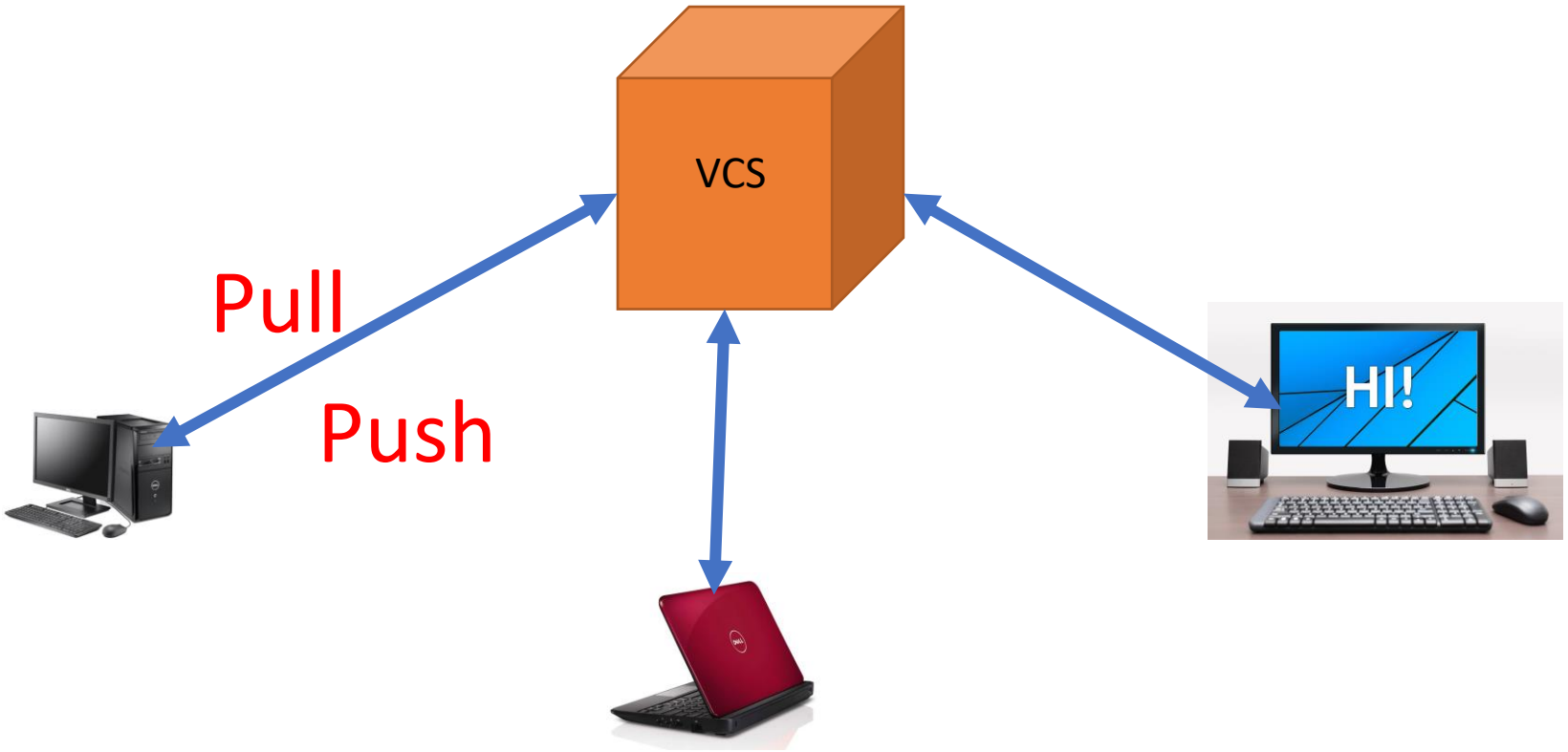
Collaboration



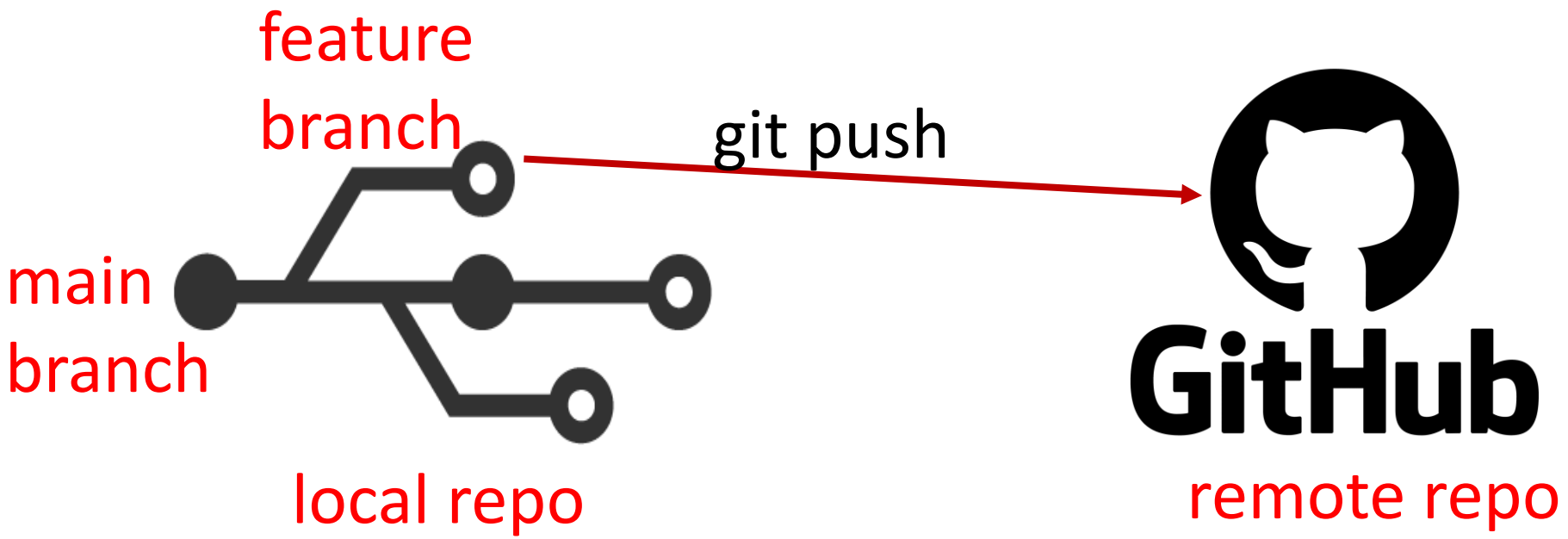
git

GitHub

How does a VCS work?




Latest
STACK
Oldest


feature
branch
git push
main
branch
GitHub
local repo
remote repo