Git

Table of Contents

Git

Git is a distributed version control system that tracks versions of files. It is often used to control source code by programmers collaboratively developing software

1) *git init:*

Create an empty Git repository or reinitialize an existing one

(***Syntax:*** git init)

```
Configuring username and mail ID,
git config --global user.name "hareesha"
git config --global user.email "Your_mail@example.com"
```

2) *git add:*

used for adding files to staging area (moves from work stage to staging area).

-versions of files not be tracked.

(***Syntax:*** git add filename)

3) *git commit:*

This will commit file from staging area to git repository (staging area to git repo)

(***Syntax:*** git commit -m "message")

4) *git status:*

used to check whether files are in work stage /in staging area/ in git repository (work space or staging area or in git repo)

(***Syntax:*** git status)

5) *git log:*

history of commits

(***Syntax:*** check → git log

remove log → rm -rf .git (remove recursively the git file)

git init (making the folder a git repo again))

6) *git rm:*

Removes files from git repository

(***Syntax:*** remove ➔ git rm filename

➔ git rm -r directory_name

commit ➔ git commit -m "message")

Steps to add any file into repo: (local)

1. Initialize the folder/repository (git init)
2. Create any file (touch g1 (or) vim g1)
3. Add file to git (git add g1)
4. Commit file to repo (git commit -m "adding g1")
5. Check status (git status)
6. Check history/log (git log)

[***Note:*** git commit and git add at same time
(***Syntax:*** git commit -am "Your commit message)]

7) *git checkout:*

switch the branches, switch the main version of code, switch tags, previous version (lets you navigate between the branches created by git branch)

(***Syntax:*** git checkout <branch_name>)

8) *git tag:*

- Tag is a name given to a set of versions of files and directories.
- Tag indicates milestone of a project; we can easily remember tag in future.

[Note: By creating tag, it saves data till files are committed when tag is getting created, it won't store further commits. Similarly, follows for next tags.]

(***Syntax:*** create ➔ git tag <tag_name>
check ➔ git tag
delete locally ➔ git tag -d <tag_name>
delete in remote ➔ git push origin: <tag_name>)

9) *git branch:*

Used for Parallel development, two people/teams will work on the same piece of code. Later, we can integrate by merging. (Git branches are effectively a pointer to a snapshot of your changes)

[Note: when we create branch, everything will be copied to that branch…. once branch is created then master and branch are individual/independent so any changes will not be copied/replicated]

(***Syntax:*** git branch <branch_name> ➔ Create the branches
git branch ➔ list the branches/check in which branch we are in
delete locally ➔ git branch -d <branch_name>
delete in remote ➔ git push origin: <branch_name>)

10) *git clone:*

git clone will bring the remote repository for the first time to the local workspace. Once we clone it, it will become non-bare repository.

(***Syntax:*** git clone url
ex.,git clone https://github.com/hareeshab/may-2024.git)

11) git pull:

It is use to bring the changes from remote repository and merges to local workspace automatically. (remote to local repo)

-It's like update repository with the latest code.

(***Syntax:*** git pull url

git pull https://github.com/hareeshab/may-2024.git)

[***Note:*** **git clone** is done for first time only, once changes are made in file then we need to do **git pull** which will update repo,

pull can be used any number of times but clone is used only once]

12) git fetch:

It brings changes from remote repository and stores in separate branch (FETCH_HEAD), we can review the changes and merge manually if it's required.

(***Syntax:*** git fetch (or) git fetch url → fetch branch will created in .git (hidden folder)

go to .git ➔ cd .git ,

list all branches/files ➔ ls)

13) git push:

Used to push changes from local workspace to remote repo.

(***Syntax:*** git push (or) git push url

git push --tag ➔ to push tags to remote repo

git push –branch (or) git push --set-upstream origin <branch_name> ➔ push branch to remote repo)

14) Bare repo:

It acts as central repo/remote repo. We can only push and pull the changes to this repo, we can't run any git operations.

(***Syntax:*** git init --bare)

15) Non-Bare repo:

It's a local repository where we can run all the git operations (add, modify, delete)

(***eg.*** git init, git add, git rm, etc)

Steps to add any file into repo: (remote from local)

1. Initialize the folder/repository (git init)
2. Create any file (touch g1 (or) vim g1)
3. Add file to git (git add g1)
4. Commit file to repo (git commit -m "adding g1")
5. git push (add/update changes in remote repo)
6. gill pull (add/update changes in local repo)

16) git merge:

- Used to merge latest changes of different branches.
- To merge we should be in base branch (branch where we want to merge) and merge branch which we want to merge into base branch.

(***Syntax:*** git merge <branch_name>
eg. git merge b1)

17) git rebase:

Similar to merging, copies and overwrites all files commits of the target branch to base branch and will be in linear order but won't create new commit ID like it used to get while using git merge.

(***Syntax:*** git rebase <branch_name>)

18) merging conflict:

Merging conflict will occur when same piece of code is changed on two different branches. When we try to merge those 2 branches, merging conflict will occur. (if we try to merge f1 from b1 to f1 in b2, and it has same content in few lines and have different content in other lines then conflict arises)

Sol to resolve merging conflict:

I don't know whose changes should I taken to the merge so I contact those two developers who modify the code on 2 different branches using a git log. They will decide whose changes we need to consider in that file so we will keep that changes and continue merging.

19) git revert:

Used to undo the changes for previous commit but does not delete the history, we can easily track who has done the revert because revert will create new commit ID for each revert.

(***Syntax:*** git revert commit_ID)

20) git reset:

Reset is used to reset the previous commit, this will delete files from git repo, staging area and also from workspace.

There will no clue that who has done the reset because history will also get deleted.

(***Syntax***: git reset commit_ID)

21) git cherry-pick:

choosing commit from one branch and applying it to another branch

(***Syntax:*** git cherry-pick -x <commit_ID>)

22) git squash:

Merge multiple commits into single commit

(***Syntax:*** git rebase -i HEAD~n) ➔ n (no of commits) variable like 1,2,3…

23) *git hooks:*

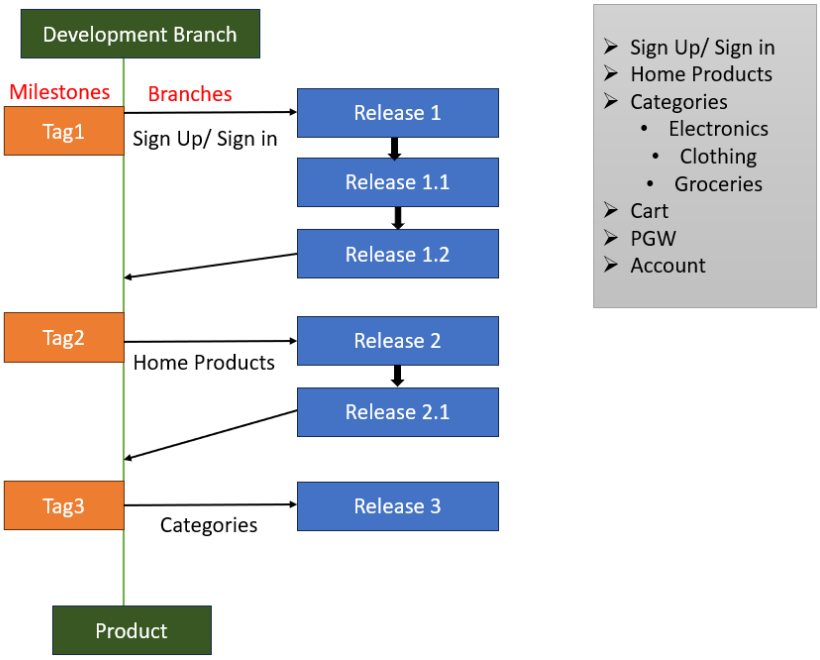
It will impose some policy before commit and after commit

pre-commit hooks: git to execute this file when we run git commit before we commit.
post-commit hooks: git will run this file after the commit.

24) *PR (pull request): (remote → GitHub)*

Pull request is when we want to merge our branch to main branch in remote repo (GitHub), once our code is prefect without any bugs and no conflicts then we can give pull request and choose reviewer as our senior official, once it's approved by them then our branch will be merged into main branch.

25) *Branching Strategy: Task/ release/ feature*



- Branches can be created for multiple reasons; here, we have created them for releases. For example, release1 is for Sign Up/Sign In, release2 is for home products, and so on.
- Tags are milestones that represent different development stages like sign in/sign up, home products, categories, etc.
- Development starts on the development branch. When the code is ready for the first release (Sign Up/Sign In), we create branch release1 from the development branch. Fixes in this branch result in versions like Release 1.1, Release 1.2, etc. Once all fixes are done (no bugs), then final release is merged back into the development branch.
- Simultaneous development continues on the development branch. Issues in release1 are fixed within release1 as it serves as a maintenance branch. Before creating release2, we merge release1 into the development branch. Then, we create release2 to ensure issues from release1 do not affect future releases.
- Similarly, followed by release 2, release 3 and so on. If there are no errors in the code then no need of merging that release branch to development branch.

26) Git Stash

- **Temporary Storage**: It saves your modified tracked files, staged changes, and untracked files to a stack of unfinished changes.
- Used for keeping files which we don't want to commit and modify it later.
- **Clean Working Directory**: After stashing, you're working directory is clean, allowing you to switch branches or pull updates without conflicts.
- **Later Retrieval**: You can later apply (git stash apply) the stashed changes back to your working directory.
  - **i) Stash file**: git stash <filename>
    command stashes all the changes in your working directory
  - **ii) Stash with a Message**: git stash -m "your message"
  - **iii) List Stashes**: git stash list
    This lists all the stashes you have saved
  - **iv) Pop Stashed Changes**: git stash pop
    Similar to git status
  - **v) To check files in stash:** git stash show
  - **vi) To bring back stash files to stages area:** git stash apply
  - **vii) To bring files to workspace:** git restore --staged <filename>

27) Git Fork

- **Definition**: A fork is a copy of a repository that resides on your GitHub account. Forking a repository creates an independent copy of the entire project.
- **Use Case**: Forking is commonly used to contribute to open-source projects or to create a personal copy of someone else's repository that you can modify freely.
- **Scope**: A forked repository is separate from the original repository. You can make changes to your fork without affecting the original repository. You can also submit pull requests to the original repository to propose changes.
- **Typical Workflow**:
  i) Fork the repository on GitHub.
  ii) Clone your forked repository to your local machine:
  git clone https://github.com/your-username/repository-name.git
  iii) Create a new branch and make changes.
  iv) Push changes to your forked repository.
  v) Create a pull request from your forked repository to the original repository.