Branching & Merging

Branching Basics
Branches are the concepts
$ cd demo
$ git status

→ The best practice is creating the feature branches or top branches from the master/ main branch and perform development and finally integrate to the master/ main branch when the feature or topic branch is stabilized.

$ git branch
→ List local branches

$ git branch -a
→ List local as well as remote branches
→ * indicates the current active branch

Creating the branch
$ git branch mynewbranch
$ git branch -a
$ git checkout mynewbranch
$ git branch -a
$ git log -- oneline -- decorate
[commit_id] (HEAD, origin/master, origin/HEAD, mynewbranch, master) {commit_message}
→ It indicates several levels associated with it.
→ HEAD, origin/HEAD are the pointers to the last commit on the respective branch both local and remote.
→ Master is the branch we are working
→ mynewbranch is the newly created branch

Since we have not made any changes yet, the branch levels point to the same commit.
→ Branches are just levels.

Renaming the branch
$ git branch -m {oldname} {newname}
$ git branch -m mynewbranch newbranch

Deleting branch
$ git branch -d newbranch
$ git branch -a

Fast forward merge
$ cd demo
$ git branch
$ git branch -a

$ git checkout -b title-change
→ Create branch & checkout
$ git status
$ git testfile1.txt
$ git status
$ git commit -am "Commit message"
→ Add the changes and commit
$ git log --oneline
$ git checkout master
$ git diff master title-change
$ git merge {name of the source branch to be merged}
$ git merge title-change
$ git log --oneline --graph --decorate
→ HEAD is pointing to the latest commit.

$ git branch
$ git branch -d title-change
$ git branch
$ git log --oneline --graph --decorate

Disable Fast Forward merges
$ git checkout -b dffmerge
$ git branch
$ vim testfile1.txt
$ git status
$ git commit -am "Disabling fast forward merge"
$ vim README.md
$ git commit -am "Some message"
$ git log --oneline --graph --decorate
$ git checkout master
$ git merge dffmerge --no-ff
→ Some Merge messages can be seen.
$ git log --oneline --graph --decorate --all
→ The graphical line is being preserved.
$ git branch -d dffmerge
$ git log --oneline --graph --decorate --all
→ All the levels can still be seen

Automatic merges
$ cd demo
$ git checkout -b simple-changes
$ vim testfile1.txt
$ git status
$ git commit -m "Adding some changes"
$ git checkout master

Before merging, let's make some changes to the files in the master as well.

$ vim README.md
$ git status
$ git commit --am "Some message"
$ git log --oneline --graph --decorate --all
$ git branch
$ git merge simple-changes -m "Merging from simple-changes branch"
$ git log --oneline --graph --decorate --all
→ The simple-changes commit is still preserved as separate commit
$ git branch
$ git branch -d simple-changes
$ git log --oneline --graph --decorate --all
→ The level simple-changes have been removed, the branch itself is intact.

Conflicting merges & Resolutions
$ cd demo
$ git status
$ git checkout -b mergeconflictpractice
$ vim README.md
$ git status
$ git commit -m "making changes to readme"
$ git status
$ git checkout master
$ vim README.md
$ git status
$ git add README.md
$ git commit -m "Adding conflicting changes on purpose for example"
$ git log --oneline --graph --decorate --all
$ git branch
$ git diff main mergeconflictpractice
$ git merge mergeconflictpractice
$ ls
$ vim README.md

→ Now we need to fix the conflict
$ git commit -m "Done with resolving conflicts"
$ git status
→ Untracked file may be visible.
$ vim .gitignore
*Add *.orig*
$ git add .gitignore
$ git commit -m "adding contents in gitignore file"
$ git branch
$ git branch -d mergeconflictpractice
$ git log --oneline --graph --decorate --all

Push changes to the GitHub
$ cd demo
$ git status
$ git branch
$ git pull oring main
$ git push origin main

Rebasing
$ cd demo
$ git status
$ git checkout -b myfeature
$ vim testfile1.txt
$ git status
$ git commit -am "some message"
$ git checkout main
$ vim README.md
$ git status
$ git commit -am "Rebase example"
$ git log --oneline --graph --decorate --all

→ We can see main and feature branches on separate lines.
In the rebasing scenario, we are still working on the feature but we also need to incorporate any changes in the master.

Rebasing master to feature branch

$ git checkout myfeature
$ git rebase {some branch}
$ git rebase master
$ git log --oneline --graph --decorate --all
$ vim README.md
$ git status
$ git commit -am "Adding another changes after the rebase"
$ git log --oneline --graph --decorate --all
$ git checkout master
$ git status
$ git diff master myfeature
$ git merge myfeature
$ git log --oneline --graph --decorate --all
$ git branch -d myfeature

Setup for rebasing conflict

$ cd demo
$ git status
→ We should be on a main branch
$ ls
$ vim testfile1.txt
→ Adding changes before rebasing conflicts.
$ git commit -am "before rebase conflicts"
$ git checkout -b bigtrouble
$ git status
$ vim testfile1.txt
$ git commit -am "fb adding some trouble to testfile"

$ git checkout master
$ git commit -am "mb adding another changes"
$ git status
$ git log --oneline --graph --decorate --all

Aborting the Rebase
$ cd demo
$ git status
$ git checkout bigtrouble
$ git branch
$ git diff master bigtrouble
$ git rebase master
$ git rebase -- abort
$ git status
$ git log --oneline --graph --decorate --all
→ Nothing changes can be seen.

Rebase conflict & Resolution
$ cd demo
$ git branch
$ git log --oneline --graph --decorate --all
$ git rebase master
$ vim testfile1.txt
$ git status
$ git add testfile.txt
$ git status
$ git rebase -- continue
$ git log --oneline --graph --decorate --all

Again make some changes,
$ vim testfile1.txt
$ git commit -am "Adding changes after rebasing"

$ git status
$ git log --oneline --graph --decorate --all
$ git checkout main
$ git merge bigtrouble

Pull with Rebase(Github)
$ cd demo
$ git status
$ git pull origin main
$ git push origin main
$ vim testfile1.txt
$ git status
$ git commit -am "Local: updating testfile"
$ git status

On the remote repository Github, modify another file
$ git status
$ git fetch
→ Fetch is a non destructive command to update the references between remote & local repositories.
$ git fetch origin main
$ git status
$ git pull -- rebase origin main
$ git status
$ git log --oneline --graph --decorate --all

Stashing

Simple stashing examples
$ cd demo
$ git status
$ vim testfile1.txt

$ git status
→ The testfile1.txt file is a work in progress and is not ready to commit this file in its current state. Support we have a requirement to modify different file So in order to save the changes, we can use the git stash command.

$ git stash
Or
$ git stash save
$ git status
$ vim README.md
$ git status
$ git commit -am "Quick fix"
$ git status
→ Now we can go back to the earlier state, to do so.
$ git stash apply
$ git status
→ Also gives the same information
$ vim testfile1.txt
→ We can continue editing the file
$ git commit -am "Done with the testfile1.txt"
$ git status
$ git stash list
→ List the pending stashes
→ The Stash work in progress(WIP), we need to remove.
$ git stash drop
→ Remove the last stash

Stashing untracked files & Using POP
$ cd demo
$ git status
$ git ls-files
→ Gives the list of files repo is tracking

$ vim testfile2.txt
$ git status
$ touch newfile.txt
→ create a new file
$ vim newfile.txt
$ git status
→ The newfile.txt is not tracked
$ git stash
→ The git stash only stash the modified files
$ git status
$ git stash apply
$ git stash drop
$ git stash list

Once we add the new file in the git staging area, git will start tracking the file.
If we don't add the file but still want to stash it,so that we can later decide the modification of the file, then we can use an extra parameter.
$ git stash -u
→ include the untracked files
$ git status
$ git stash list
$ vim README.md
$ git commit -am "Some message"
$ git status
$ git stash pop
$ git status
$ rm newfile.txt
$ git commit -am "Update the file"
$ git status

Managing Multiple Stashes

$ cd demo
$ git status
$ ls
$ vim testfile.txt
$ git status
$ git stash save "Simple changes"
$ vim README.md
$ git stash save "Readme changes"
$ vim testfile2.txt
$ git stash save "Changes in testfile2.txt"
$ git stash list

Note: The last stash is indexed 0
$ git stash show stash@{1}
→ stash@{1} is called reflog syntax, it allows you to reference the specific stash to show.
$ git status
$ git stash list
$ git stash apply stash@{1}

To apply stash
$ git status
$ git stash list
$ git stash drop stash@{1}
$ git stash list
$ git stash clear
→ To clear the stash list
$ git stash list

Stashing into a Branch
$ cd demo
$ git status

$ git stash list
$ vim testfile2.txt
$ vim testfile3.txt
$ git status
$ git add testfile2.txt
$ git status
$ touch new.md
$ git status
→ We now realized that this changes are for feature branch not for main so let's stash
$ git stash -u
$ git status
→ To apply the stash to the new branch
$ git stash branch newchanges
→ A new branch "newchanges" is created, switched, stash is applied & the stash is dropped
$ git stash list
$ git status
$ rm new.md
$ git add .
$ git commit -m "Some message"
$ git checkout main
$ git merge newchanges
$ git branch -d newchanges
$ git branch

Syncing the changes to the remote GitHub
$ git pull origin main
$ git push origin main

Git Tagging

Simple Tag Example/ Lightweight Tags

$ cd demo
$ git status
$ git log --oneline --graph --decorate --all

We may have made a lot of changes to the repository and we want to mark significant events or milestones in the repository. That can be accomplished by using tagging support.

Tag are the labels that can be applied to any commit in history.

$ git tag {tag_name}
$ git tag myTag

This kind of tag is called lightweight tag

$ git log --oneline --graph --decorate --all
→ We can see the new item added.

$ git tag -- list
→ Lists the tags

$ git show myTag
→ We can use the name of the tag in other git commands as well.

$ git tag --delete myTag
→ To delete tags

$ git tag -- list
$ git log --oneline --graph --decorate --all

→ The myTag should now be removed from the log as well.

Annotated Tags

$ git status
→ It is similar to the lightweight tag except it has little extra information. It usually has what's equivalent to the commit message but for tags.

$ git tag -a v-1.0
-a represents annotated tag

$ git tag -- list
$ git log --oneline --graph --decorate --all
$ git show v-1.0

Comparing tags

$ cd demo
$ git status
$ git tag -- list
$ vim testfile1.txt
$ git add testfile1.txt
$ git commit -m "add testfile1.txt"
$ git log --oneline --graph --decorate --all
$ git tag -a v1.1
$ vim testfile2.txt
$ git commit -am "add testfile2.txt"
$ git commit --amend
→ To amend the committed message.
$ git tag v1.2 -m "tag message"
$ git tag –list
$ git log --oneline --graph --decorate --all

$ git diff v-1.0 v-1.2

Tagging a specific commit(Previous commit)
$ cd demo
$ git status
$ git log --oneline --graph --decorate --all
$ git tag -a v-0.9-beta {commit_id}
$ git log --oneline --graph --decorate --all
$ git tag -a v-0.8-alpha {commit_id}
$ git log --oneline --graph --decorate --all

Updating tags / Updating an existing Tag
$ cd demo
$ git status
$ git log --oneline --graph --decorate --all
$ git tag -a v-0.8-alpha -f {commit_id}
→ -f means force

Remote Tagging, using tags with Github
$ cd demo
$ git status
$ git tag -- list
$ git log --oneline --graph --decorate --all
$ git push origin v-0.9-beta

let's check on the UI.

$ git push origin v-1.1

To push all total tags
$ git push origin main --tags

To delete a tag from Github

$ git tag --list

$ git push origin :v-0.8-alpha

→ Deletes the v0.8-alpha