GIT AND GITHUB NOTES

Git:-

Git is an open-source distributed version control system. It is designed to handle minor to major projects with high speed and efficiency. It is developed to co-ordinate the work among the developers. The version control allows us to track and work together with our team members at the same workspace.

Git is foundation of many services like GitHub and GitLab, but we can use Git without using any other Git services. Git can be used privately and publicly.

Git was created by Linus Torvalds in 2005 to develop Linux Kernel. It is also used as an important distributed version-control tool for the DevOps.

Git is easy to learn, and has fast performance. It is superior to other SCM tools like Subversion, CVS, Perforce, and ClearCase.

Features of Git

Open Source

Git is an open-source tool. It is released under the GPL (General Public License) license.

Scalable

Git is scalable, which means when the number of users increases, the Git can easily handle such situations.

Distributed

One of Git's great features is that it is distributed. Distributed means that instead of switching the project to another machine we can create a clone of entire repository also instead of just having one central repo that you send changes to every user has their own repo that contains the entire commit history of project. We do not need to connect to the remote repo the change is just stored on our local repo if necessary we can push changes to remote repository

----

Security

Git is secure. It uses the SHA1 (Secure Hash Function) to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout. It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.

Speed

Git is very fast, so it can complete all the tasks in a while. Most of the git operations are done on the local repository, so it provides a huge speed. Also, a centralized version control system continually communicates with a server somewhere.

Performance tests conducted by Mozilla showed that it was extremely fast compared to other VCSs. Fetching version history from a locally stored repository is much faster than fetching it from the remote server. The core part of Git is written in C, which ignores runtime overheads associated with other high-level languages.

Git was developed to work on the Linux kernel; therefore, it is capable enough to handle large repositories effectively. From the beginning, speed and performance have been Git's primary goals.

Supports non-linear development

Git supports seamless branching and merging, which helps in visualizing and navigating a non-linear development. A branch in Git represents a single commit. We can construct the full branch structure with the help of its parental commit.

Branching and Merging

Branching and merging are the great features of Git, which makes it different from the other SCM tools. Git allows the creation of multiple branches without affecting each other. We can perform tasks like creation, deletion, and merging on branches, and these tasks take a few seconds only. Below are some features that can be achieved by branching.

We can create a separate branch for a new module of the project, commit and delete it whenever we want.

We can have a production branch, which always has what goes into production and can be merged for testing in the test branch.

We can create a demo branch for the experiment and check if it is working. We can also remove it if needed.

The core benefit of branching is if we want to push something to a remote repository, we do not have to push all of our branches. We can select a few of our branches, or all of them together.

Data Assurance

The Git data model ensures the cryptographic integrity of every unit of our project. It provides a unique commit ID to every commit through a SHA algorithm. We can retrieve and update the commit by commit ID. Most of the centralized version control systems do not provide such integrity by default.

Staging Area

The Staging area is also a unique functionality of Git. It can be considered as a preview of our next commit, moreover, an intermediate area where commits can be formatted and reviewed before completion. When you make a commit, Git takes changes that are in the staging area and make them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the changes.

Although, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.

Another feature of Git that makes it apart from other SCM tools is that it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory.

Maintain the clean history

Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.

GitHub is an immense platform for code hosting. It supports version controlling and collaboration and allows developers to work together on projects. It offers both distributed version control and source code management (SCM) functionality of Git. It also facilitates collaboration features such as bug tracking, feature requests, task management for every project.

Essential components of the GitHub are:

Repositories

Branches

Commits

Pull Requests

Git (the version control tool GitHub is built on)

Advantages of GitHub

GitHub can be separated as the Git and the Hub. GitHub service includes access controls as well as collaboration features like task management, repository hosting, and team management.

The key benefits of GitHub are as follows.

It is easy to contribute to open source projects via GitHub.

It helps to create an excellent document.

You can attract the recruiter by showing off your work. If you have a profile on GitHub, you will have a higher chance of being recruited.

It allows your work to get out there in front of the public.

You can track changes in your code across version.

----

features of GitHub

GitHub is a place where programmers and designers work together

They collaborate contribute and fix bugs together

it host plenty of open source projects and code of various

some of its significant features as follows

collabration

Integrated issue and bug tracking

Graphical representation of branches

Git repositories hosting

Project management

Team management

Code hosting

Track and assign tasks

Conversations

GitHub

Git

It is a cloud-based tool developed around the Git tool.

It is an online service that is used to store code and push from the computer running Git.

It is dedicated to centralize source code hosting.

It is managed through the web.

It provides a desktop interface called GitHub desktop GUI.

It has a built-in user management feature.

It has built in user management feature

It has market place for tool configuration

It is a distributed version control tool that is used to manage the programmer's source code history.

Git tool is installed on our local machine for version controlling and interacting with online Git service.

It is a command-line utility tool.

The desktop interface of Git is called Git GUI.

It is dedicated to version control and code sharing.

The desktop interface of Git is called Git GUI.

It does not provide any user management feature

It has a minimal tool configuration feature.

GitHub

it was launched in 2008 it is a git repository hosting service it is free for public repositories and paid for private repositories

it has gist's(a way to share code snippets)

we cannot attach any file to any issue

it has a fast interface

in GitHub we can decide the read or write access of a user to a reposirtory

it is the largest repository hosting service it contains approximate 100+ Mil repositories

Gitlab

it was launched as a project in 2011 as an alternative to the available git repository hostings service

it is free for both public and private repositories

it does not have gist's

we can attach any file to any issue

comparatively it has slow interface

in Gitlab we can set and modify user permissions according to their roles

it has lesser projects than GitHub

GitHub

It has user friendly and fast interface

it is limited to the git repository only

it facilities with graph pulse contributors commits code frequency members of it

it is free for public repositories and paid for private repositories

GitHub comes with a lot of feature and allows you to create your workflows

we cannot make private repo on free accounts

Bitbucket

it has a slick and clean interface which provides a professional view

it is not limited to just git it also supports other version control system like mercurial but id does not support svn

It assists with REST APIs to build third-party applications which can be used in any development language.

It is free for both private and public repositories. But we can have a maximum of five members for a private repository.

Bitbucket provides a more built-in option for flexibility.

We can create an unlimited private repository for up to five users.

----

svn cpmmand	Git command	Git behaviour
status	status	Report the state of working tree
add	add	Required to add each path before making a commit
commit	commit	store prepared changes in local revision history
rm,delete	rm	prepare the path for deletion in next commit
move	mv	prepare relocated content for next commit
checkout	clone	acquire the entire history of project locally for the first time
	branch	create local context for commit
	merge	join branch histories and changes to working here
	log	no network required
	push	upload commit histories and change to working here
	pull	download and integrate GitHub repositories history with local on
	fetch	download GitHub repository with no other action

submodule

git submodule add https://github.com.gothubraining/example

git commit -m "adding new submodule"

The submodule add command adds a new file called .gitmodules along with subdirectory containing new files from example submodule both are added to your index and you simply need to commit them the submodule's history remains independent of the parent project

git subtree add -- prefix=example-submodule https://github.com

The subtree command adds a subdirectory containing all the files from example-submodule the most common practice is to use the --squash option to combine the subproject history into single commit which is grafted onto the existing tree of the parent project you can omit the --squas option ot maintain all of the history from the designated branch of the subproject

https://github.com/githubtraining/example-submodule main --squash

viewing a diff of the subproject

to view a diff on the submodule

#show changes to the submodule to commit

git diff example -submodule

#show online log of new commits in the submodule

git diff --submodule example-submodule

#show changes to the file in the submodules

git diff --submodule=diff

cloning a repository with a subproject

submodule

to clone a repository along with submodule

```
git clone --recurse-submodule URL
```

if you forgot --recurse-submodules you can clone and initialize all submodules

```
git submodule update --init --recursive
```

Adding --recursive is only required if any submodule itself has submodules

pulling in superproject updates

Submodule

by default the submodule repository is fetched but not updated when you run git pull in the subproject you need to use git sub module update or add the --recurse-submodule flag to pull;

```
git pull
git submodule ypdate --init --recursive
#or ,in one step git
git pull --recurse-submodules
```

--init is required if the superproject added new submodules and --recursive is needed if any submodule itself has submodules if ever the superproject changes the url of the submodule a separate command is required if ever the superproject changes the url of the submodule a separate command is required

```
#copy the new url to your local config
git submodule sync --recursive
```

-recursive is only needed if any submodules itself had submodules

Changing Branches

Submodule

By default, the submodule working tree is not updated to match the commit recorded in the superproject when changing branches.

You need to use git submodule update, or add the —recursive-submodules flag to switch

```
git swith <branch>
git submodule update --recursive
# or , in one step (Git >= 2.14)
git switch --recurse-submodules <branch>
```

pulling in subproject updates

submodule

```
#update the submodule repository
git submodule update --remote
#record the changes in the superproject
git commit -am "update submodule"
```

if you have more than one submodule can add the path ti the submodule at the end of the git submodule update --remote command to specify which subproject to subproject to update.

By default, git submodule update —remote will update the submodule to the latest commit on the main branch of the submodule remote.

You can change the default branch for further calls with:

#git >=2.22

git submodule set -branch other-branch

#or

git -f .gitmodule submodule.example-submodule.branch

making changes to a subproject

in most cases it is considered best practices to make changes in a separate clone of the subproject repository and pull them into parent project when this is not practical

Submodule	Subtree
Access the submodule directory and create a branch changes will be committed on the present project bench	No special command required
cd example-submodule git switch -b branch-name main	It is possible to create commits mixing changes to the subproject and the parent project, but this is generally discouraged
Changes require two commits one in the subproject repository and one in the parent repository. Don’t forget to push in both the submodule and the superproject	

Pushing Changes to the Subproject Repository

Submodule

While in the submodule directory:

Plain Text

```
git push
```

Or while in the parent directory:

Plain Text

```
git push --recurse-submodules=on-demand
```

Subtree

Plain Text

```
git subtree push --prefix=example-submodule https://github.com/githubtraining
```

Helpful Configs for Submodules

Always show the submodule log when you diff:

Plain Text

```
git config --global diff.submodule log
```

Show a short summary of submodule changes in your git status message:

Plain Text

```
git config --global status.submoduleSummary true
```

Make push default to --recurse-submodules=on-demand:

Plain Text

```
git config --global push.recurseSubmodules on-demand
```

Make all commands (except clone) default to --recurse-submodules if they support the flag (this works for git pull since Git 2.15):

Plain Text

```
git config --global submodule.recurse true
```

Advanced Git operations:

Branches: A branch is separate "version" of a repo with its own history it is not a separate copy it exist within repo where it was created.

```
        o --little feature

      o---o-----o------o  --> main

           o--- big feature
```

Two branches create from the main branch each with their own commit history

```
The                   main tip
                    |
       o---o-----o------o  --> main
        '       |
       ^--common base o----feature tip
```

the main and feature branch have a common base but different tip

```
                 main tip
                   |
       o---o-----o------o----o    new merge commit
```

```
        |      |     |
    common base o---o---o----feature tip
```

making your collaborators are working on code simultaneously each once can create their own branch pushing commits as they work no merge conflicts would occur in the main branch during that time saving the hassle of res

Testing code in software and web development

for example many R packages are downloadable directly from the GitHub repo lets say the developer is testing out change that she want to push to GitHub so that other collaborators can see them int that situation it is ideal to create to

Branchworkflow

1. create a new branch locally

2.switch to the new branch locally

3.push the local branch to remote repo

4.Make some changes stage commit and push (repeat as needed)

5.merge the branch back into main branch

fork

To fork a repo is to copy a repo from someone else's account into your account so that you can start your own project based on the existing repo without having to push changes to the original owners repo in git jargon a fork is just a clone with a different remote origin if you fork a repo it is probably going to be one where you aren't collaborating directly with its creator the creator may continue to make updates adding commits to their versions history while you do the same. A good example would be if you are making a GitHub page and you want to use a premade template you would fork the page and you want to use a premade template repo add your own content you would likely never push changes to the template itself.

However if you are working on a project that you forked from someone else's repo and you do want them to incorporate your changes into their version of the repo you would make a pull request which we will cover later in the lesion

forking a repo

To fork a repo in GitHub just go to the repo page and click fork in the upper right hand corner

Pull request

A pull request is just that a request that the owner of a repo pull your changes incorporating them into his or her own repo. You can make a pull request to any public repo on GitHub. even you don't have push access to the repo. in the contrast if you are a collaborator on a repo who has write access you can push commits without making pull request anyway instead of pushing directly this is a more respectful way to contribute to a project because it gives the repo owner a chance to review you changes and approve them

Basic pull request workflow

1.The pull request fork someone else's repo and clones it locally

2.s/he makes some changes,stages,commits and pushes

3. on GitHub s/he creates a pull request comparing the changes you just made with the original owners repo

4. the original owner creates a new branch and pulls the pr into that branch.

5. The owner task test the code potentially adding additional changes and committing them

6. if the code is good the owner accepts the pr by merging it into his or her main branch and pushing

GitHub issues

Issues are a useful feature of GitHub.com that can greatly improve your workflow.The user interface is fairly easy to figure out.this is just a plug to encourage everyone to make them a part of their workflow

Uses for issues

-pointing out a bug or feature request in someone elses repo

-making a to do list for yourself

-managing projects by assigning task to collaborator

-maintaining a record of rich info about your work

-troubleshooting problems with code by searching existing issues

formatting issues

you can reference users commit pull request and other issues int the text of your issues and they will automatically be linked

to reference users use the @sign followed by their name

to reference commits use the first 7 digit of the commit's hash it will look something like 5d1001b

to reference pull request and other issues use the #sign followed by the number id of the pull request or issue they go up sequentially starting at #1

issue labels and assignees

you can put labels on issues to tag them by topic

you can also assign issues to collabrators on your project to designate who is responsible for resolving the issue example issues assigned to different collabrators

closing resolved issues

once an issue is resolved you can close it will no longer appear as an outstanding issue but all closed issues are still archived as part of the repo ion GitHub.

Trouble shooting with issues

if you run into an issue with an R package or other software it is often helpful to go to the repo page for that software and search the issues there this can be a more targeted way to find a solution for your problem compared to searching google to stackoverflow.

Merge conflicts and how to handle them in git

what are merge conflict?

why do merge conflcits occur

types of merge conflicts

creating a merge conflict

handling the merge congflict

what are merge conflicts?

A merge conflict happens when git is unable to automatically reconcile differences in code between two commits.This typically occurs during a merge operations where changes from different branches are combined.

why do merge conflicts occur?

merge conflicts usually occur in following scenarios

1. simultaneous edits: two developers modify the same line of code in diff branches

2.conflicting changes: A file is deleted in one branch and modified in another

3.complex merges: when multiple branches are being merged with changes scattered across various files and lines

types of merge conflicts

while starting the merge:if there are changes in either the working directory or staging area while merging then git will fail to start the merge this happens bcos the pending changes could be overridden by the commits that are being merged.this is the error message provided by git when this type of merge conflicts happens

error:entry filename not update.cannot merge.(changes in working directory) or error entry filename would be overwritten by merge cannot merge changes in staging area

this type of conflict can be resolved either by doing git stash save any message to describe what is saved (stashes away any changes in your staging area and working directory in separate index) or git checkout file throws out your changes and then the merge can be completed

during the merge this occurs bcos you have commited changes that are in conflict with someone elses commited changes git will do its best to merge the files and will leave things for you to resolve manually in the files it lists this is the error message provided by git when this type of merge conflict happens :

CONFLICT(content):merge conflict in <file name>

Automatic merge failed; fix conflicts and then commit the results.

To show a simple example of how a merge conflict can happen, we can manually trigger a merge conflict from the following set of commands in any UNIX terminal / GIT bash :

Step 1: Create a new directory using the mkdir command, and cd into it.

Step 2: initialize it as a new Git repository using the git init command and create a new text file using the touch command.

Step 3: Open the text file and add some content in it, then add the text file to the repo and commit it.

Step 4: Now, its time to create a new branch to use it as the conflicting merge. Use git checkout to create and checkout the new branch.

Step 5: Now, overwrite some conflicting changes to the text file from this new branch.

Step 6: Add the changes to git and commit it from the new branch.

With this new branch: new_branch_for_merge_conflict we have created a commit that overrides the content of test_file.txt

step 7:again checkout the master branch and this time append some text to the test_file.txt

Step 8: add these new changes to the staging area and commit them.

Step 9: Now for the last part, try merging the new branch to the master branch and you will encounter the second type of merge conflict.

handling the merge conflict

As we have experienced from the proceeding example, Git will produce some descriptive output letting us know that a CONFLICT has occurred. We can gain further insight by running the git status command. This is what we will get after running the git status command:

```
On branch masterYou have unmerged paths.  (fix conflicts and run "git commit")  (use "git merge --abort" to abort the merge)Unmerged paths:  (use "git add <file>..." to mark resolution)    both modified:   test_file.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

On opening the test_file.txt we see some “conflict dividers”. This is the content of our test_file.txt :

```
<<<<<<< HEAD

Adding some content to mess with it later

Append this text to initial commit
```

=======

Changing the contents of text file from new branch

>>>>>>> new_branch_for_merge_conflict

The ======= line is the “center” of the conflict. All the content between the center and the <<<<<<< HEAD line is content that exists in the current branch master which the HEAD ref is pointing to. Alternatively, all content between the center and >>>>>>> new_branch_for_merge_conflict is content that is present in our merging branch.

To resolve our merge conflict, we can manually remove the unnecessary part from any one of the branches, and only consider the content of the branch that is important for further use, along with removing the “conflict dividers” from our file. Once the conflict has been resolved we can use the git add command to move the new changes to the staging area, and then git commit to commit the changes.