CHITKARA UNIVERSITY

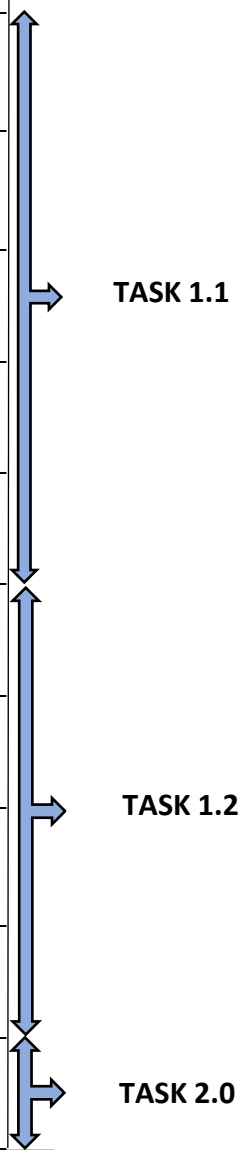
SOURCE CODE MANAGEMENT (CS181)

Task (1.1+1.2 +Project)

Submitted by –
Dhruv Jain
2110990443
G8-B

Submitted to –
Dr. Monit Kapoor

INDEX



Introduction

What is GIT?

Git is a source code management technology used by DevOps. Git is a piece of software that allows you to track changes in any group of files. It is a free and open-source version control system that may be used to efficiently manage small to big projects.

What is GITHUB?

GitHub is a version management and collaboration tool for programming. It allows you and others to collaborate on projects from any location.

What is the difference between GIT and GITHUB?

Git	GitHub
1. It is a software	1. It is a service
2. It is installed locally on the system	2. It is hosted on Web
3. It is a command line tool	3. It provides a graphical interface
4. It is a tool to manage different versions of edits, made to files in a git repository	4. It is a space to upload a copy of the Git repository
5. It provides functionalities like Version Control System Source Code Management	5. It provides functionalities of Git like VCS, Source Code Management as well as adding few of its own features

What is Repository?

A repository stores all of your project's files, as well as the revision history for each one. Within the repository, you may discuss and monitor your project's progress.

What is Version Control System (VCS)?

Version Control Systems are the software tools for tracking/managing all the changes made to the source code during the project development. It keeps a record of every single change made to the code. It also allows us to turn back to the previous version of the code if any mistake is made in the current version.

Without a VCS in place, it would not be possible to monitor the development of the project.

Types of VCS

- Local Version Control System
- Centralized Version Control System
- Distributed Version Control System

I. **Local Version Control System**: Local Version Control System is located in your local machine. If the local machine crashes, it would not be possible to retrieve the files, and all the information will be lost. If anything happens to a single version, all the versions made after that will be lost.

II. **Centralized Version Control System**: In the Centralized Version Control Systems, there will be a single central server that contains all the files related to the project, and many collaborators checkout files from this single server (you will only have a working copy). The problem with the Centralized Version Control Systems is if the central server crashes, almost everything related to the project will be lost.

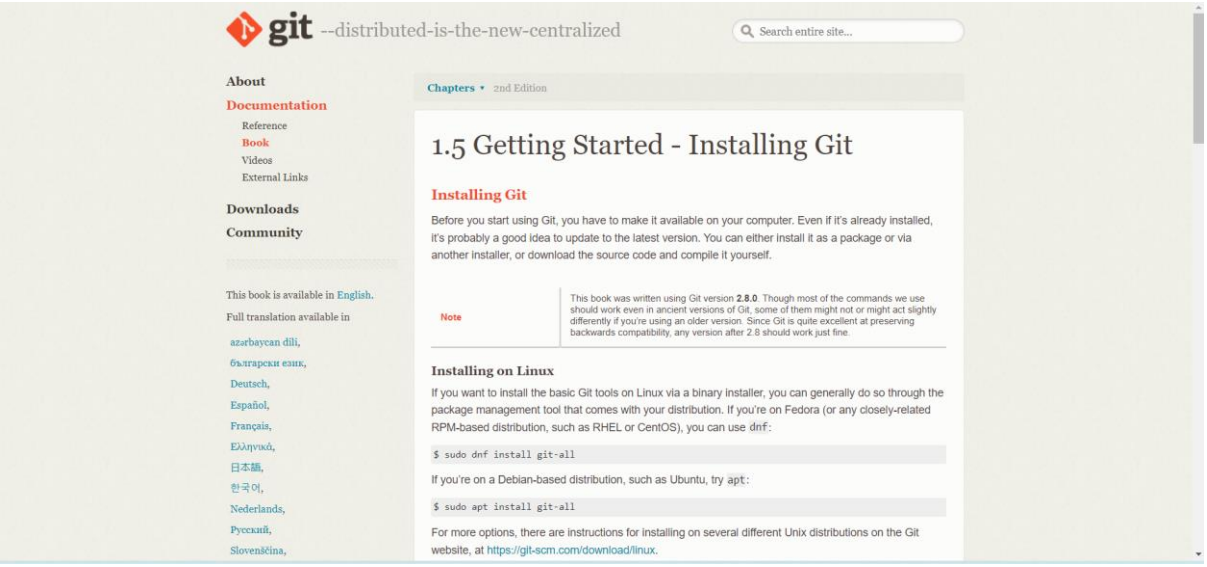
Distributed Version Control System: In a distributed version control system, there will be one or more servers and many collaborators similar to the centralized system. But the difference is, not only do they check out the latest version, but each collaborator will have an exact copy of the main repository on their local machines. Each user has their own repository and a working copy. This is very useful because even if the server crashes we would not lose everything as several copies are residing in several other computers.

Task 1.1

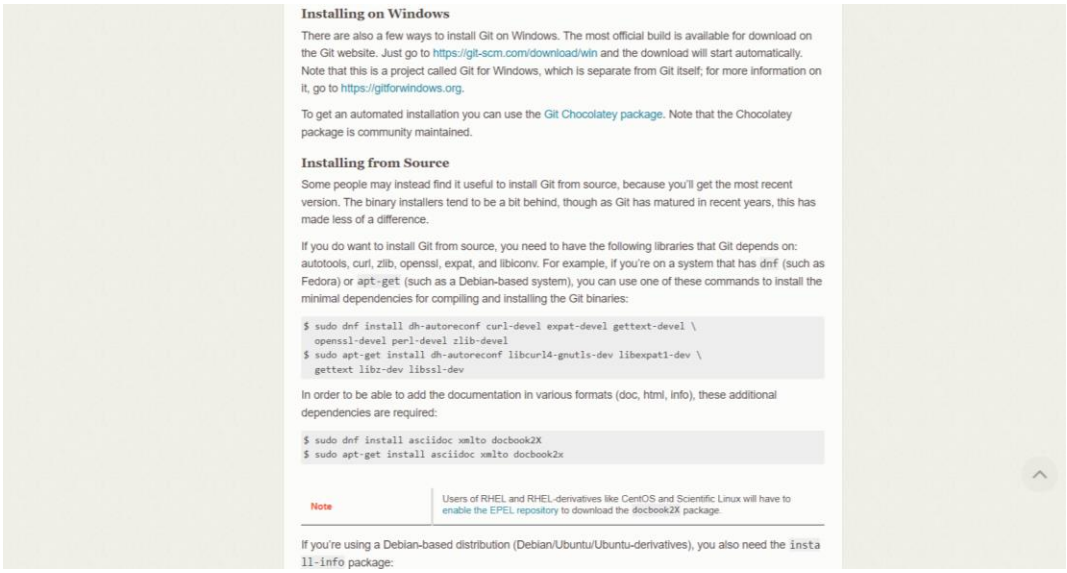
Experiment No. 01

Aim: Setting up of Git Client

✧ For git installation on your system, go to https://git-scm.com/book/en/v2/Getting-Started-Installing-Git.



✧ Choose the operating system by clicking on it. I'll choose the Windows operating system.



Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to https://git-scm.com/download/win and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to https://gitforwindows.org.

To get an automated installation you can use the Git Chocolatey package. Note that the Chocolatey package is community maintained.

Installing from Source

Some people may instead find it useful to install Git from source, because you'll get the most recent version. The binary installers tend to be a bit behind, though as Git has matured in recent years, this has made less of a difference.

If you do want to install Git from source, you need to have the following libraries that Git depends on: autotools, curl, zlib, openssl, expat, and libiconv. For example, if you're on a system that has `dnf` (such as Fedora) or `apt-get` (such as a Debian-based system), you can use one of these commands to install the minimal dependencies for compiling and installing the Git binaries:

```
$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \
  openssl-devel perl-devel zlib-devel
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
  gettext libz-dev libssl-dev
```

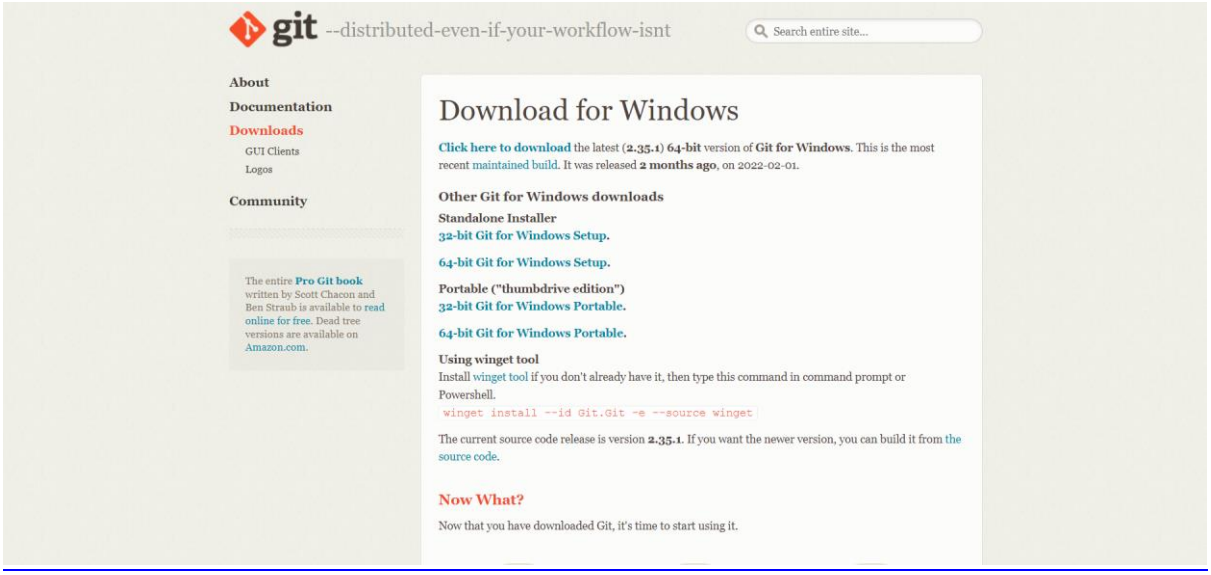
In order to be able to add the documentation in various formats (doc, html, info), these additional dependencies are required:

```
$ sudo dnf install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

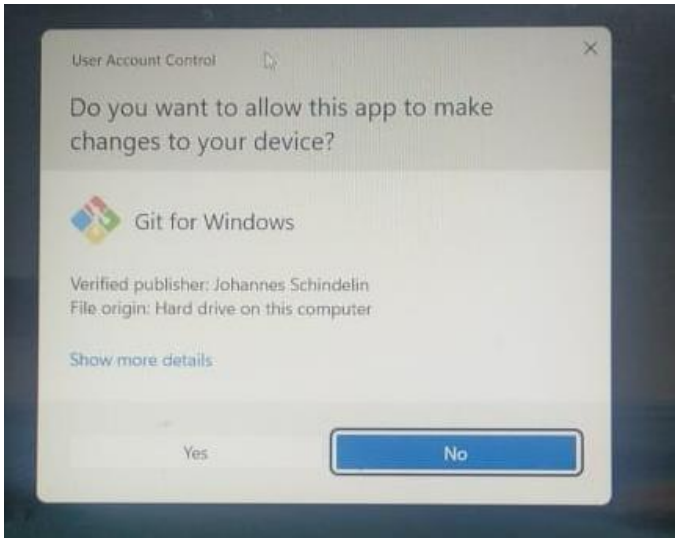
Note Users of RHEL and RHEL-derivatives like CentOS and Scientific Linux will have to enable the EPEL repository to download the `docbook2X` package.

If you're using a Debian-based distribution (Debian/Ubuntu/Ubuntu-derivatives), you also need the `installinfo` package:

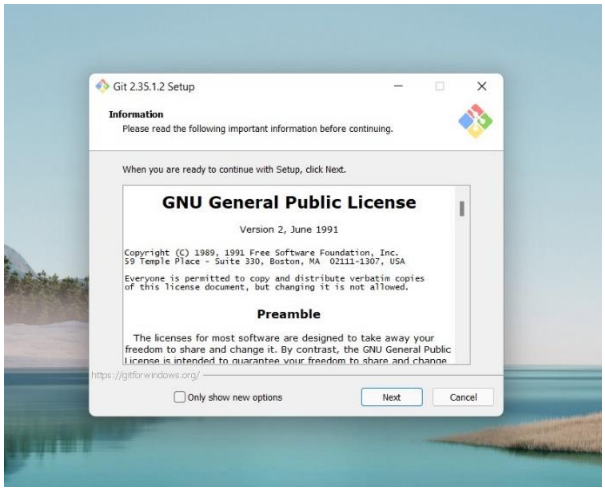
✧ Select the CPU for your system now. (I choose to 64-bit Git for Windows Setup.)



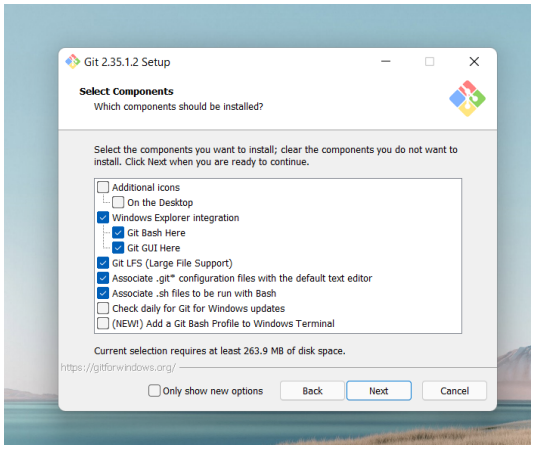
✧ Then, open the Git in Download folder. We will be asked if you want to enable this program to make modifications to your PC once you launch it. Select YES.

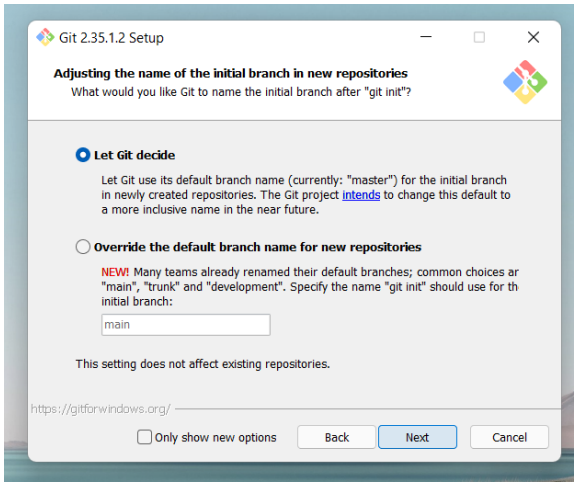


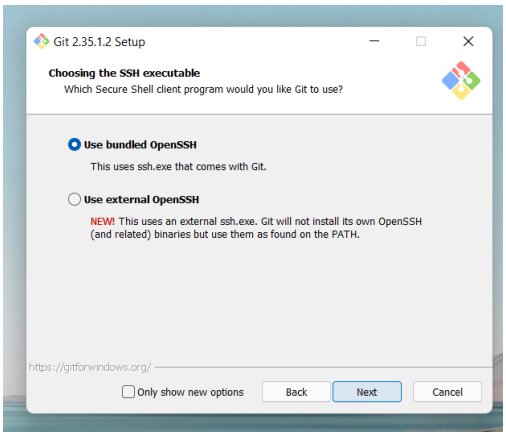
✧ The below window will appear. Click on Next.

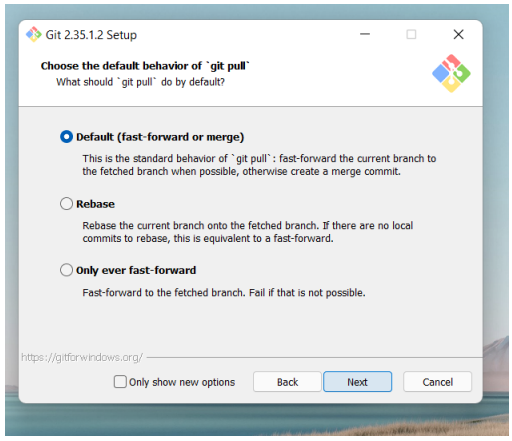


✧ Continue clicking on next few times more.

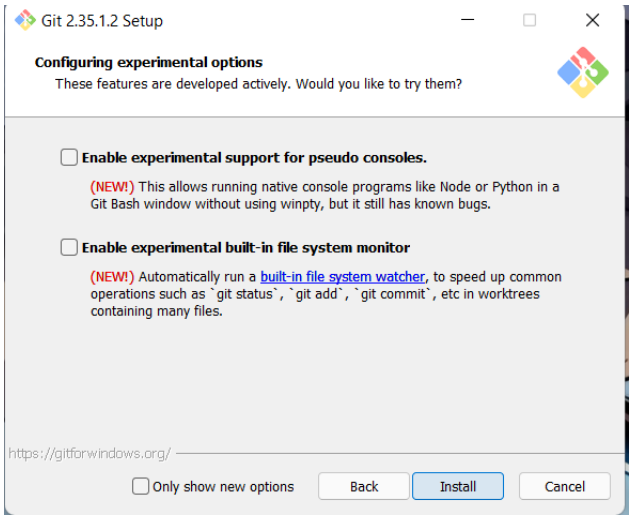


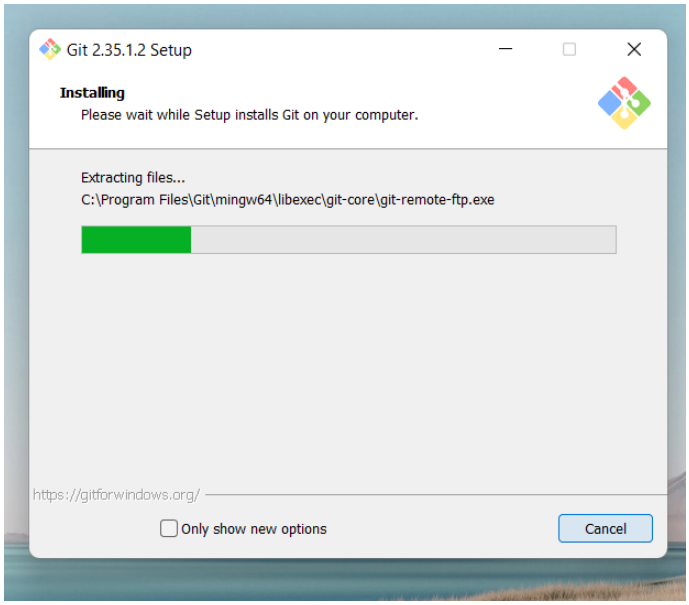




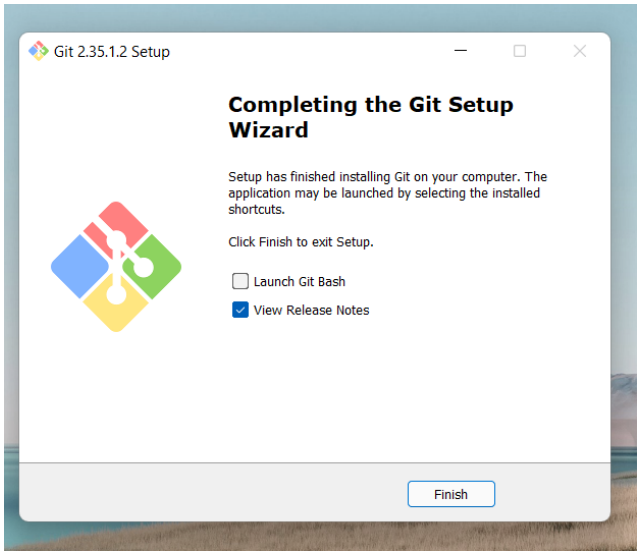


✧ Now select the Install option.





✧ After completion of installing, click on Finish after the installation is finished.



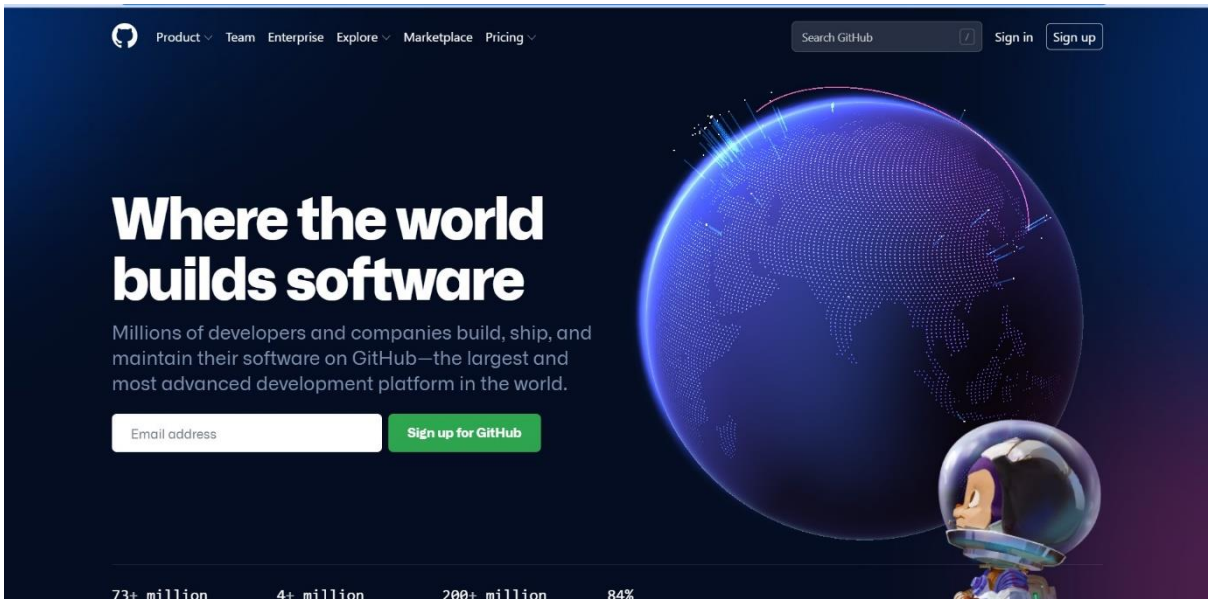
The installation of the git is finished and now we have to setup git client and GitHub account.

Experiment No. 02

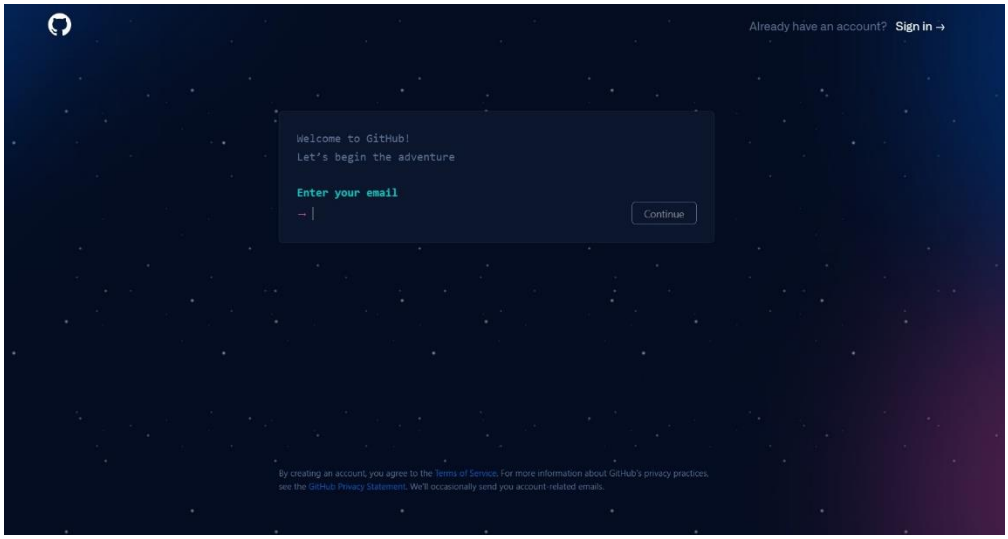
Aim: Setting up GitHub Account

- Open your web browser search http://github.com .

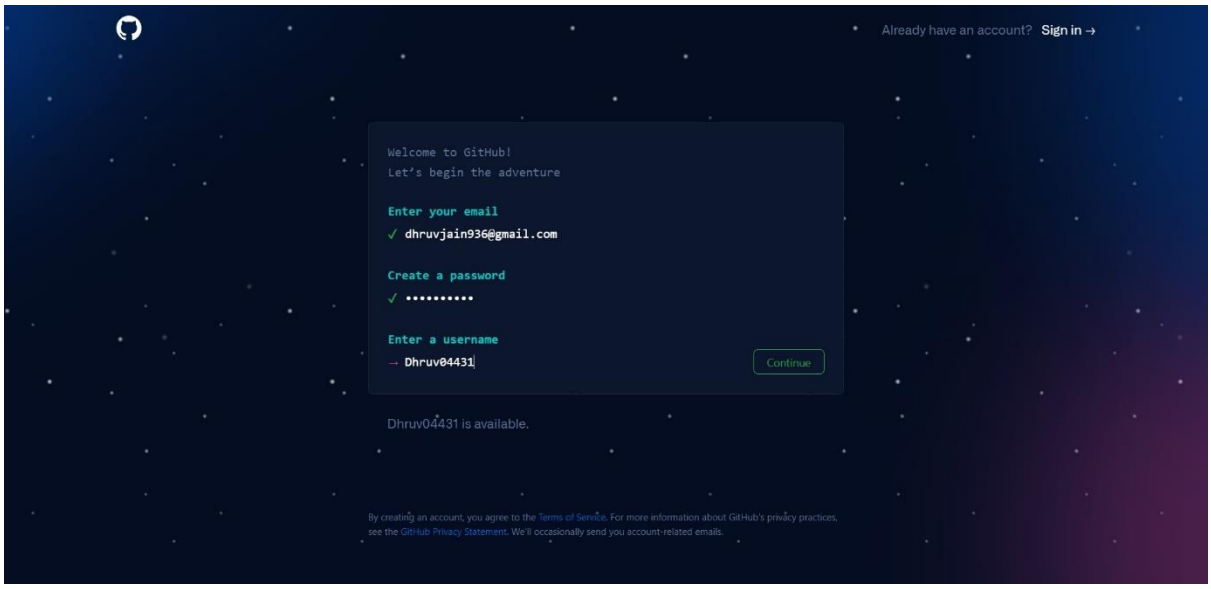
- Click on Sign Up on the upper-right conner of desktop window.



- After clicking on "Sign Up," a new page will appear where we must enter your email address for your account.

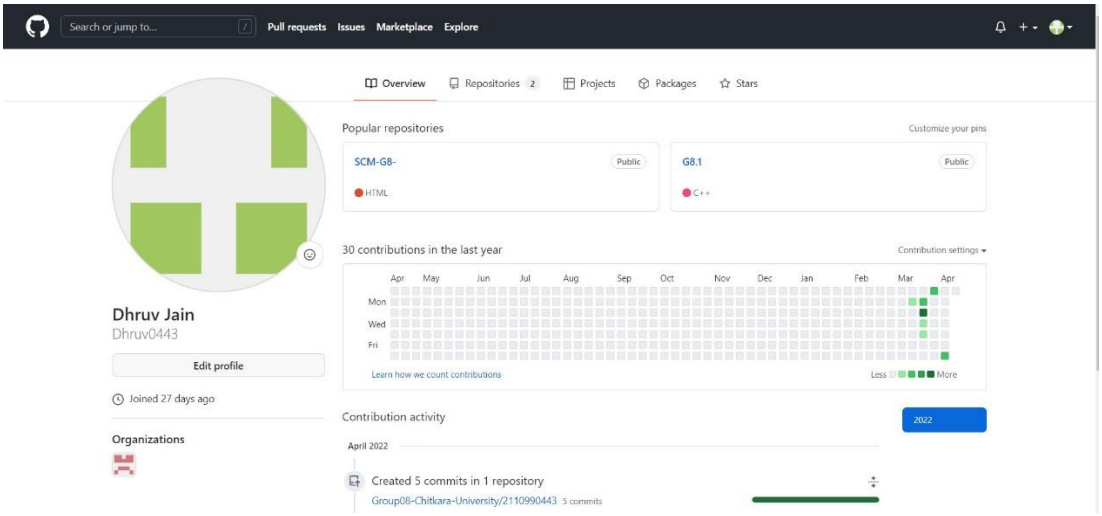


✧ Now type in the password we want to use for your GitHub account. Then you'll be prompted to enter your username.



✧ Now Click on Create Account.

✧ Verify the email and our Github account is ready.



Experiment No. 03

Aim: Program to Generate logs

✧ First of all create a local repository using Git. For this, you have to make a folder in your device, right click and select "**Git Bash Here**". This opens the Git terminal. To create a new local repository, use the command "**git init**" and it creates a folder **.git**.

```
DELL@DESKTOP-VRD4NDB MINGW64 /d/Dhruv (master)
$ git init
Initialized empty Git repository in D:/Dhruv/.git/

DELL@DESKTOP-VRD4NDB MINGW64 /d/Dhruv (master)
$
```

✧ When we use GIT for the first time, we have to give the user name and email so that if I am going to change in project, it will be visible to all.

For this, we use command →

"git config --global user.name *Name*"
"git config --global user.email *email*"

Some Important Commands:

git add -A **[** For add all the files in staging area. **]**

git commit -m "write any message" **[** For commit the file **]**

```
DELL@DESKTOP-VRD4NDB MINGW64 /d/Dhruv (master)
$ vi abc.cpp

DELL@DESKTOP-VRD4NDB MINGW64 /d/Dhruv (master)
$ git add abc.cpp
warning: LF will be replaced by CRLF in Dhruv/abc.cpp.
The file will have its original line endings in your working directory

DELL@DESKTOP-VRD4NDB MINGW64 /d/Dhruv (master)
$ git commit -m"Changing Output"
[master ea7cef1] Changing Output
 1 file changed, 1 insertion(+), 1 deletion(-)
```

✧ **git log:** The git log command displays a record of the commits in a Git repository. By default, the git log command displays a commit hash, the commit message, and other commit metadata.

```
DELL@DESKTOP-VRD4NDB MINGW64 /d (master)
$ git log
commit 67697cd9caecf8ce2f7ec5e1d797d1924d8400d8 (HEAD -> master, file/master)
Author: Dhruv0443 <dhruv0443.be21@chitkara.edu.in>
Date:   Sat Apr 9 22:50:38 2022 +0530

    REMOVING Conditions from loops

commit 748eee1f51f067252168888389fd9a8703eae4f9 (htmlfile/master)
Author: Dhruv0443 <dhruv0443.be21@chitkara.edu.in>
Date:   Sat Apr 9 22:45:27 2022 +0530

    Changing background colour

commit d64708be2be5a7f88961fc99ca0200410268d348
Author: Dhruv0443 <dhruv0443.be21@chitkara.edu.in>
Date:   Sat Apr 9 22:41:49 2022 +0530

    FLex Making in html
```

Experiment No. 04

Aim: Create and visualize branches

✧ **Branching:** A branch in Git is an independent line of work(a pointer to a specific commit). It allows users to create a branch from the original code (master branch) and isolate their work. Branches allow you to work on different parts of a project without impacting the main branch.

Let us see the command of it:
Firstly, add a new branch, let us suppose the branch name is activity1.
For this use command →

- **git branch name [**adding new branch**]**
- **git branch [**use to see the branch's names**]**
- **git checkout *branch name* [**use to switch to the given branch**]**

```
Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (master)
$ git branch
  acitivity3
  activity1
  activity2
* master

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (master)
$ git branch act1

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (master)
$ git checkout act1
Switched to branch 'act1'

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (act1)
$ git branch
  acitivity3
* act1
  activity1
  activity2
  master
```

In this you can see that firstly 'git branch' shows only one branch in green colour but when we add a new branch using 'git branch act1', it shows 2 branches but the green colour and star is on master. So, we have to switch to act1 by using 'git checkout act1'. If we use 'git branch', now you can see that the green colour and star is on act1. It means you are in activity1 branch and all the data of master branch is also on act1 branch. Use "ls" to see the files.

Now add a new file in activity1 branch, do some changes in file and commit the file.

```
Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (master)
$ touch contact.html

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (activity1)
$ git add -A

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (activity1)
$ git commit -m "committing contact.html"
[activity1 af7b1d9] committing contact.html
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 contact.html

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (activity1)
$ ls
'Activity 1 file.txt'   contact.html  'wallpaperflare.com_wallpaper (1).jpg'
 comments.txt           index.txt     'wallpaperflare.com_wallpaper (2).jpg'

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (activity1)
$
```

If we switched to master branch, ‘contact.html’ file is not there. But he file is in activity1 branch.

```
Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (activity1)
$ git checkout master
Switched to branch 'master'

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (master)
$ ls
 OOPScalcu.py                filewrite.py    randomGame.py         'wallpaperflare.com_wallpaper (1).jpg'
 Playsound.py                findinfile.py   sample.txt            'wallpaperflare.com_wallpaper (2).jpg'
 comments.txt                index.txt       song.mp3              'wallpaperflare.com_wallpaper (4).jpg'
'factorial TrailingZero.py'  os.py           temp1.txt             'wallpaperflare.com_wallpaper (5).jpg'
 file.py                     pratice.py      try_except_final.py   'wallpaperflare.com_wallpaper (7)g.jpg'

Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (master)
$
```

- To add these files in master branch, we have to do merging. For this firstly switch to master branch and then use command →

  git merge branchname [use to merge branch**]**

```
Amit@LAPTOP-6T2OQEHA MINGW64 ~/OneDrive/Desktop/G3 scm (master)
$ git log
commit 87ba6dc52876a0cda3d9397ec902c91e42dd79e0 (HEAD -> master)
Merge: 898faf3 af7b1d9
Author: XxFiEnDxX <amitkumargdgps@gmail.com>
Date:   Tue Mar 29 01:09:42 2022 +0530

    Merge branch 'activity1'
```

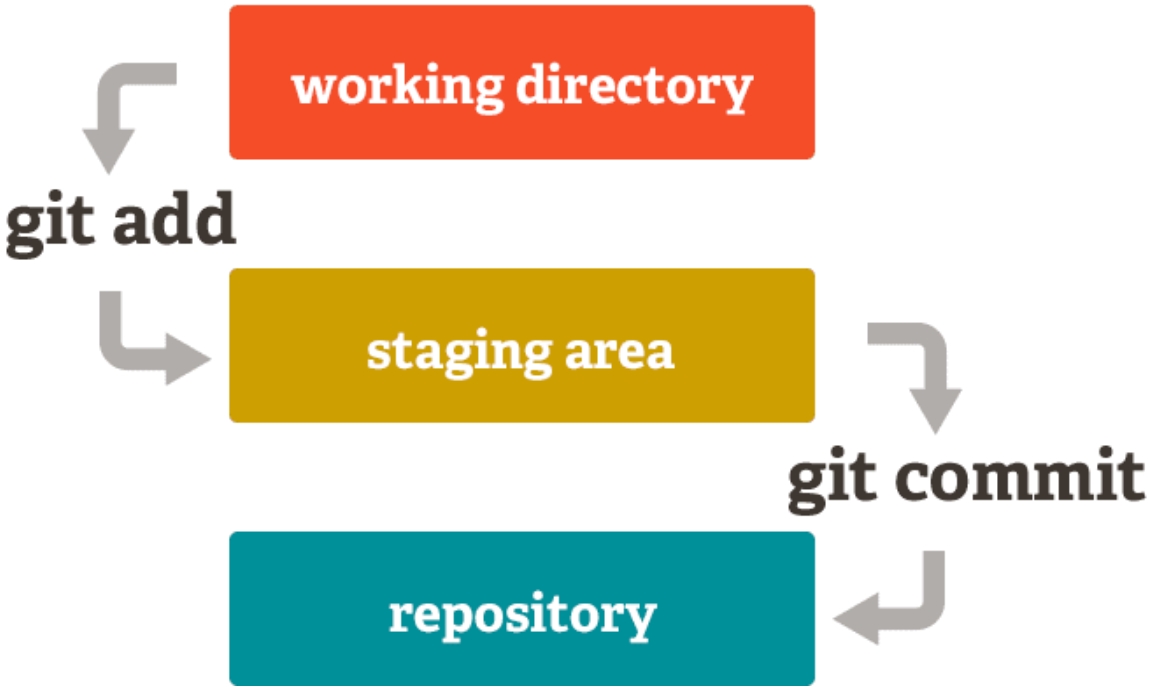
Experiment No. 05

Aim: Git lifecycle description

Stages in GIT Life Cycle:

When a directory is made a git repository, there are mainly 3 states which make the essence of Git Version Control System. The three states are –

- Working Directory
- Staging Area
- Git Directory



- **Working Directory:** Whenever we want to initialize our local project directory to make it a git repository, we use the ***git init*** command. After this command, git becomes aware of the files in the project although it doesn't track the files yet. The files are further tracked in the staging area.

```
git init
```

- **<u>Staging Area</u>:** Now, to track the different versions of our files we use the command ***git add***. We can term a staging area as a place where different versions of our files are stored. ***git add*** command copies the version of your file from your working directory to the staging area.

```
// to specify which file to add to the staging area
git add <filename>
// to add all files of the working directory to the staging area
git add .
```

- **<u>Git directory(repository)</u>:** Now since we have all the files that are to be tracked and are ready in the staging area, we are ready to commit our files using the ***git commit*** command. Commit helps us in keeping the track of the metadata of the files in our staging area. We specify every commit with a message which tells what the commit is about.

```
git commit -m <Commit Message>
```

Task 1.2

Experiment No. 06

Aim: Add Collaborators on GitHub Repository

In GitHub, We can invite other GitHub users to become collaborators to our private repositories (which expires after 7 days if not accepted, restoring any unclaimed licenses). Being a collaborator, of a personal repository you can pull (read) the contents of the repository and push (write) changes to the repository. You can add unlimited collaborators on public and private repositories (with some per day limit restrictions). But, in a private repository, the owner of the repo can only grant write-access to the collaborators, and they can't have the read-only access.

GitHub also restricts the number of collaborators we can invite within a period of 24 hours. If we exceed the limit, then either we have to wait for 24-hours or we can also create an organization to collaborate with more people.

Actions that can be Performed By Collaborators

Collaborators can perform a number of actions into someone else's personal repositories, they have gained access to Some of them are,

- Create, merge, and close pull requests in the repository.
- Publish, view, install the packages.
- Fork the repositories.
- Make the changes on the repositories as suggested by the Pull requests.
- Mark issues or pull requests as duplicate.
- Create, edit, and delete any comments on commits, pull requests, and issues in the repository.
- Removing themselves as collaborators on the repositories. • Manage releases in the repositories.

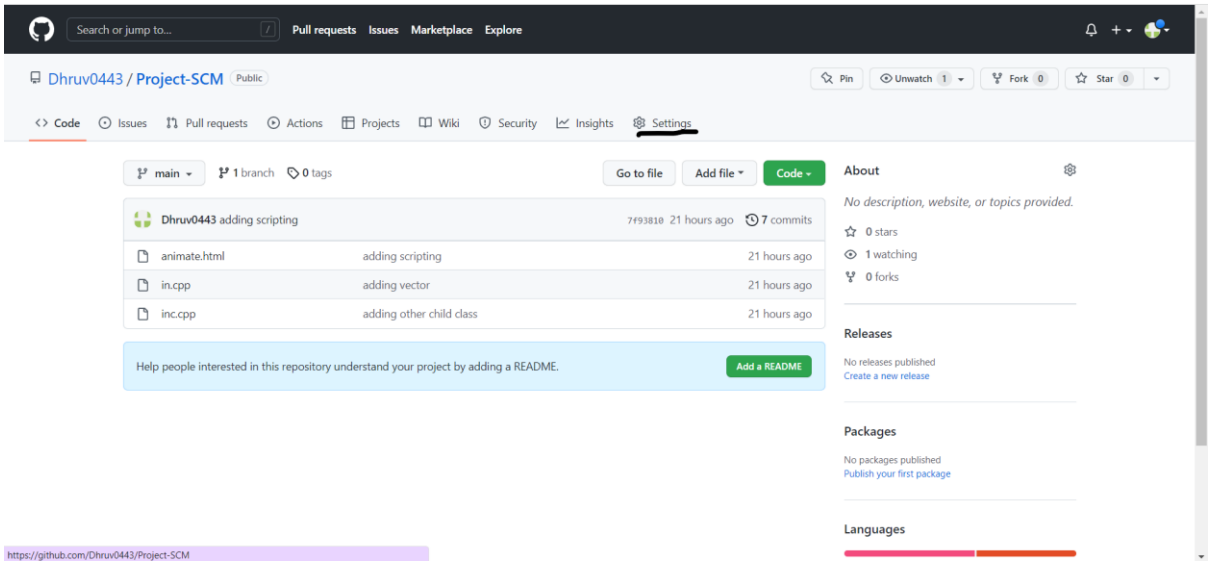
Inviting Collaborators to your personal repositories

Follow the steps below to invite collaborators to your own repository (public or private).

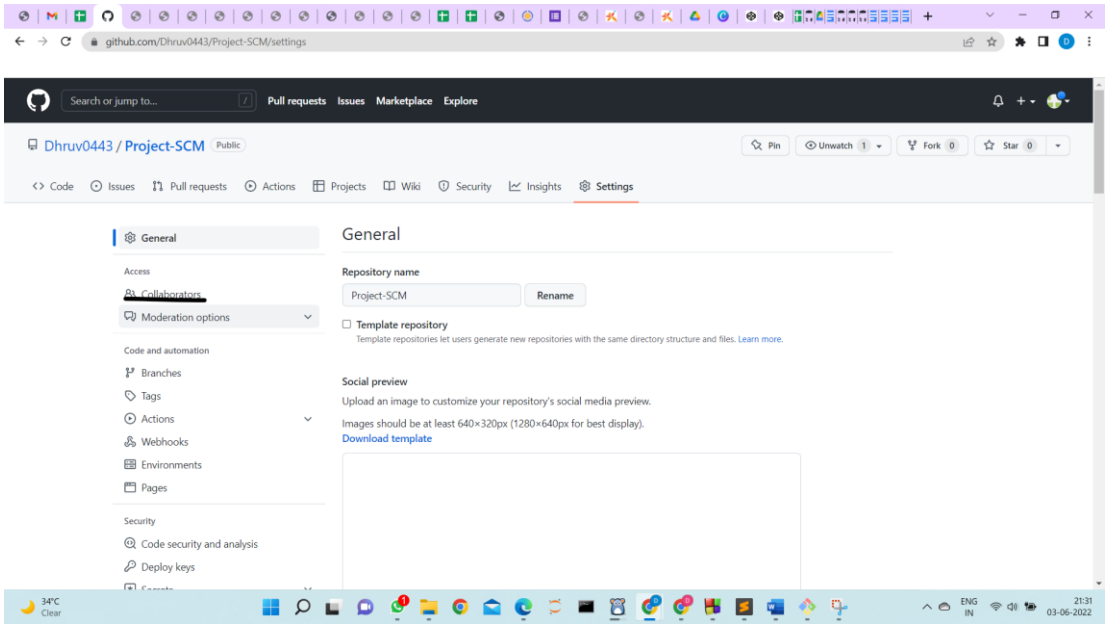
Step 1: Get the usernames of the GitHub users you will be adding as collaborators. In case, they are not on GitHub, ask them to sign in to GitHub.

Step 2: Go to your repository (intended to add collaborators)

Step 3: Click into the Settings.



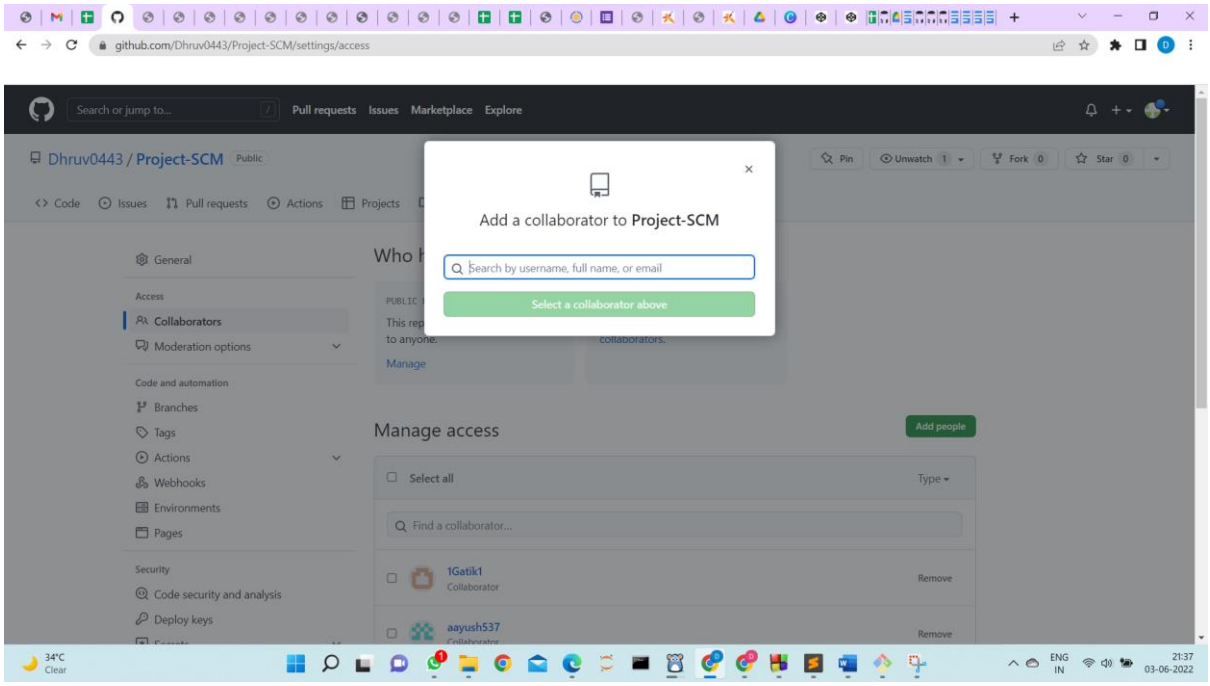
Step 4: A settings page will appear. Here, into the left-sidebar click into the Collaborators.



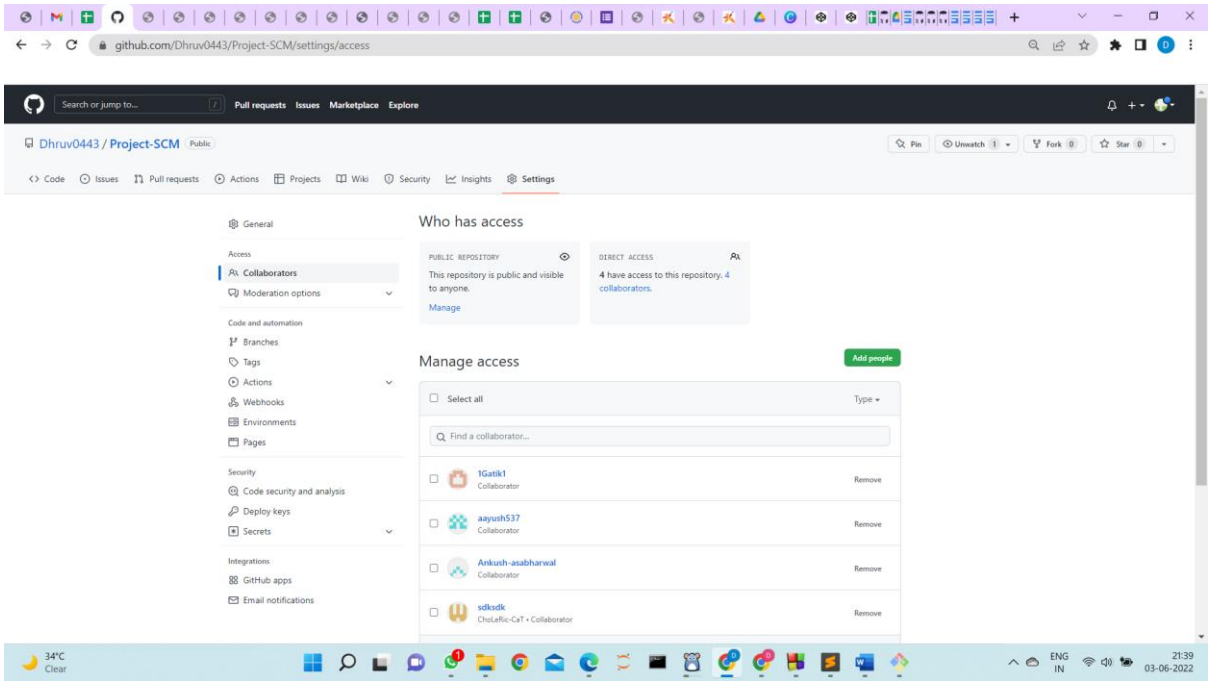
Step 5: Then a confirm password page may appear, enter your password for the confirmation.

Step 6: Next, click on Add People.

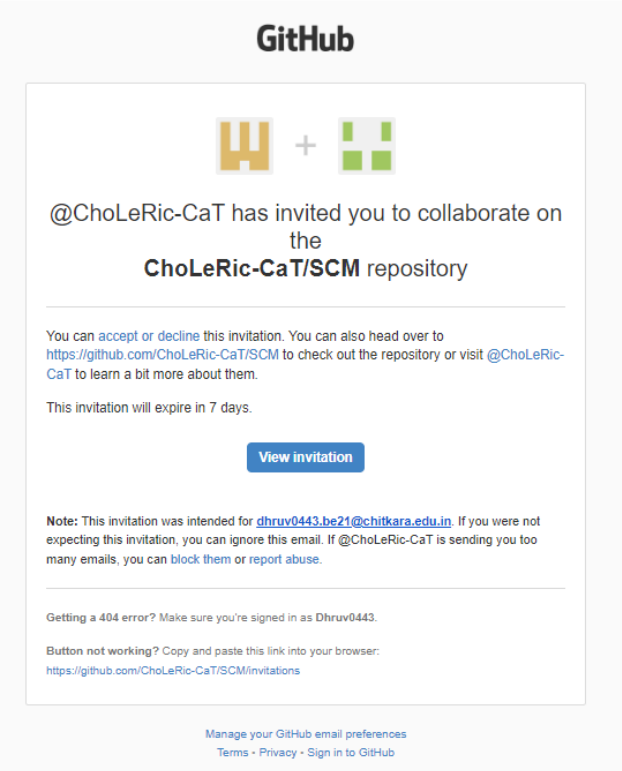
Step 7: Then a search field will appear, where you can enter the username of the ones you want to add as collaborator.



Step 8: After selecting the people, add them as collaborator.



Step 9: After sending the request for collaboration to that person whom we wanted as our collaborator for our project will get a email from the Team leader (by GitHub). He/she has to open its Email to view the invitation.



Step 10: After Clicking the View invitation, He/she will be redirected to the GitHub Page for accepting the invitation sent by the team leader.

Step 11: We are done adding a single collaborator. Now, they will get a mail regarding invitation to your repository. Once they accept, they will have collaborator access to your repository. Till then it will be in pending invitation state. You can also add more collaborator and delete the existing one as depicted below.

Removing collaborator permissions from a person contributing to a repository

Similar to the above steps, go to Your Repository -> Settings -> Manage Access -> Remove (on the right side of collaborator username).

Experiment No. 07

Aim: Fork and Commit

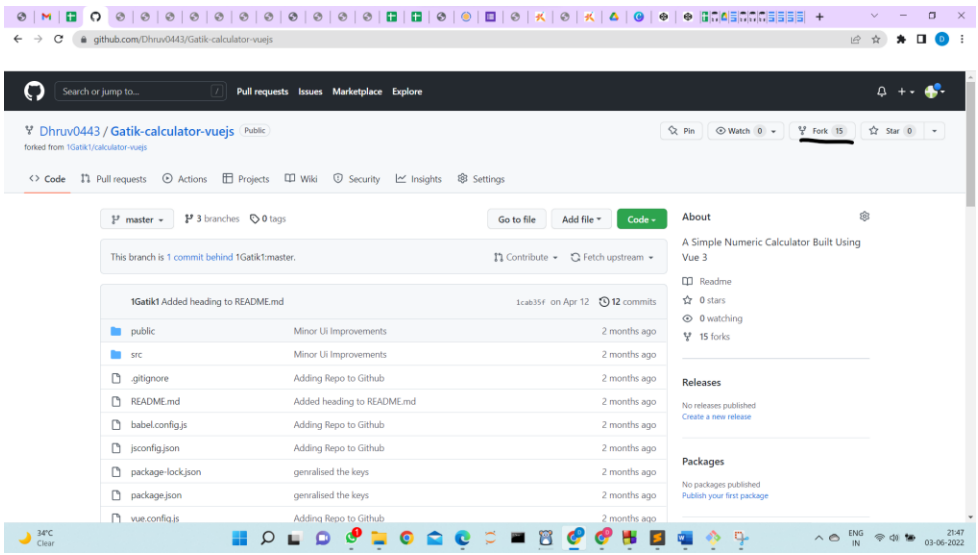
What does Forking a Repository mean and Why is it used?

Forking a repository means creating a copy of the repo. When you fork a repo, you create your own copy of the repository on your GitHub account. This is done for the following reasons:

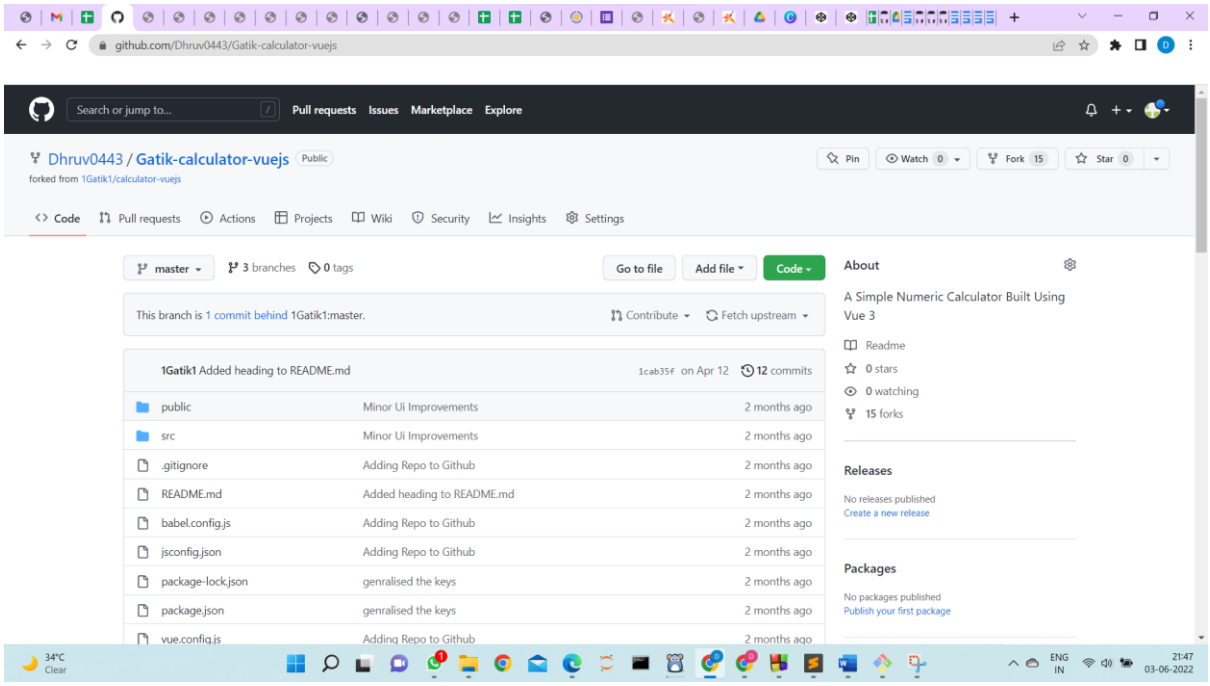
1. You have your own copy of the project on which you may test your own changes without changing the original project.
2. This helps the maintainer of the project to better check the changes you made to the project and has the power to either accept, reject or suggest something.
3. When you clone an Open Source project, which isn't yours, you don't have the right to push code directly into the project.

For these reasons, you are always suggested to FORK. Let's have a screenshot walkthrough of the whole process. When getting started with a contribution to Open Source Project, you have been advised to first FORK the repository(repo). But what is a fork?

You must have seen this icon on every repository in the top right corner. Now, this button is used to Fork the repo. But again, what is a fork or forking a repository in GitHub as shown in the below media as follows:

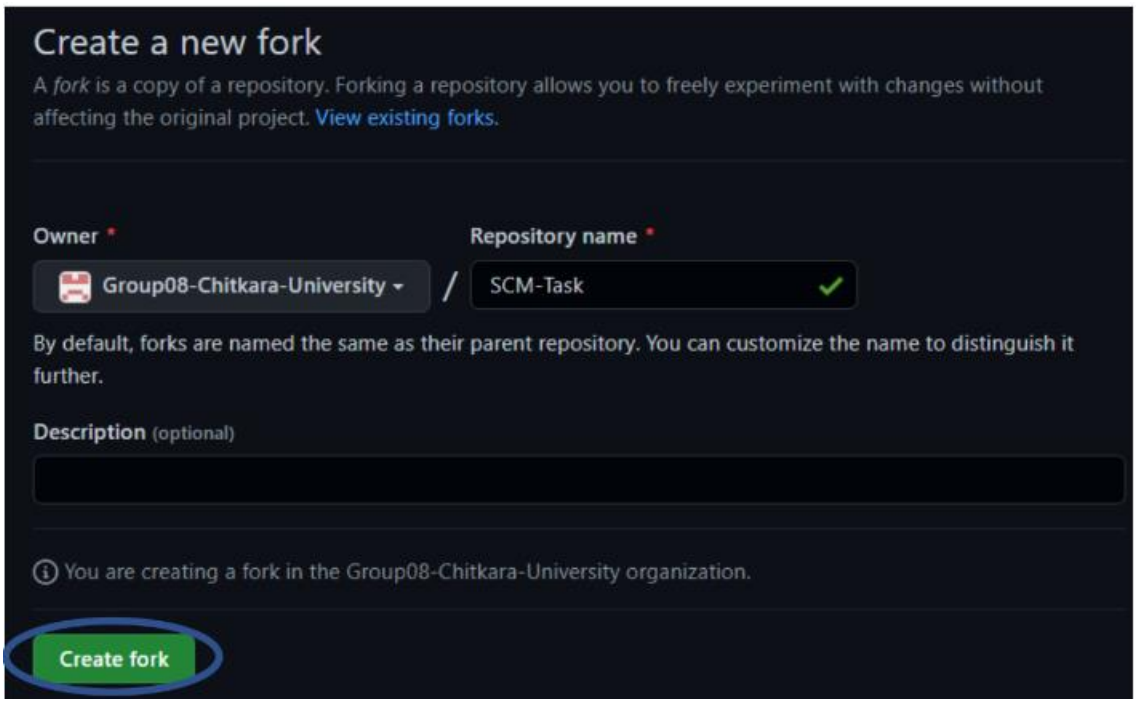


Step 1: Go to **SCM-Task** official repository.

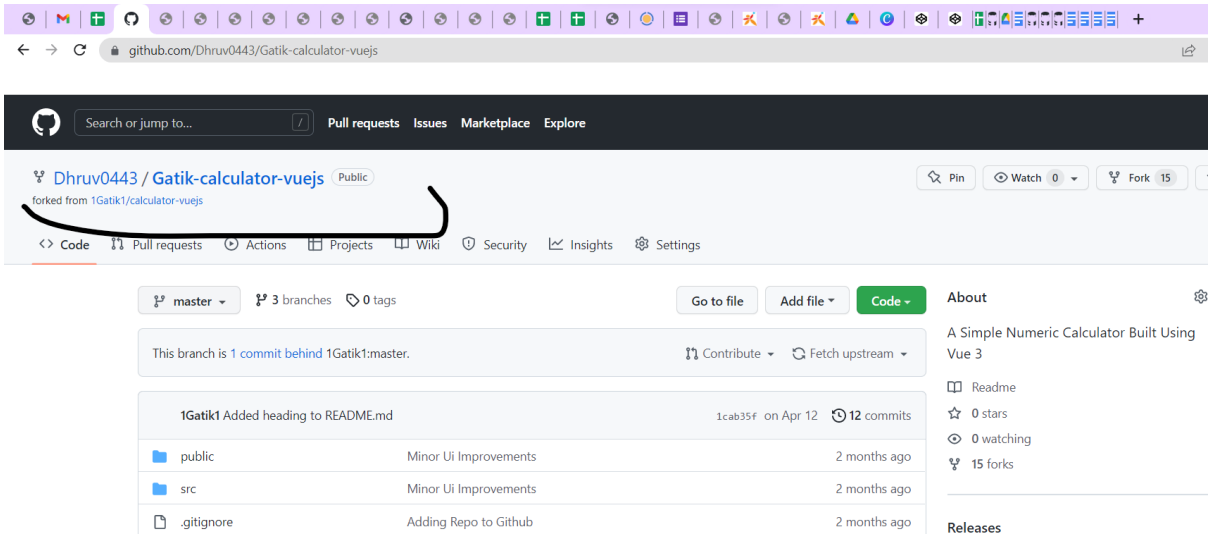


Step 2: Find the Fork button on the top right corner.

Step 3: Click on **Fork**. You will see this screen.



Step 4: Now you have your own copy of the repository. But how can we confirm for which do refer to below visual aid as follows:



We can make my changes here and then make a Pull Request to the maintainers of the project. Now it is in their hand if they will accept or reject your changes to the main project.

After this, we will see how to work in the forked repositories which were forked by the collaborators. If the collaborator tries to change in his forked repository, it will not affect the main repository.

What is COMMIT in GitHub?

Commit is like a snapshot of your repository. These commits are snapshots of your entire repository at specific times. You should make new commits often, based around logical units of change. Over time, commits should tell a story of the history of your repository and how it came to be the way that it currently is. Commits include lots of metadata in addition to the contents and message, like the author, timestamp and more.

It is similar to saving a file that's been edited, a commit records changes to one or more files in your branch. Git assigns each commit a unique ID, called a SHA or hash, that identifies:

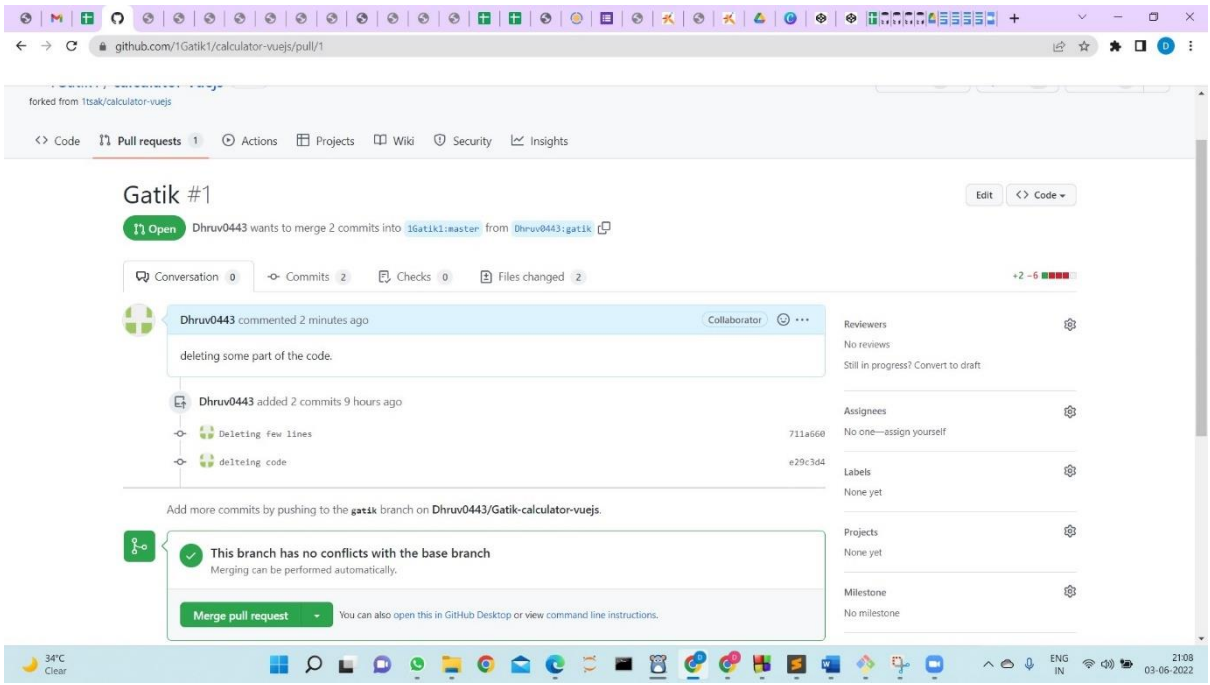
- The Specific Changes
- When the Changes were made

• Who created the Change

When you make a commit, you must include a commit message that briefly describes the changes, you can also add a co-author on any commits you collaborate on.

We can see that the file has been edited and now it will be committed in this forked repository.

We can see the Commit history of the Repository (Forked Repository).



Experiment No. 08

Aim- Merge and resolve Conflicts due to owner's - Activity and Collaborator's Activity

How does the merge conflict occur?

A merge conflict occurs when both the owner and collaborator change the same lines in the same file without first pulling the changes that the other has made. This is most easily avoided by good communication about who is working on various sections of each file, and trying to avoid overlaps. But sometimes it happens, and *git* is there to warn you about potential problems. And git will not allow you to overwrite one person's changes to a file with another's changes to the same file if they were based on the same version. The main problem with merge conflicts is that, when the Owner and Collaborator both make changes to the same line of a file, git doesn't know whose changes take precedence. You have to tell git whose changes to use for that line.

How to resolve the merge conflicts?

Mergetool - **git mergetool** is used to run one of several merge utilities to resolve merge conflicts. It is typically run after ***git merge***

```
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git init
Reinitialized existing Git repository in C:/taskfiles1.2/.git/

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ touch code.cpp

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ cat code.cpp
#include <iostream>
using namespace std;

int main(){
    cout<<"A+B=MORNING"
}
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ cat code.cpp
#include <iostream>
using namespace std;

int main(){
    cout<<"A+B=MORNING";
    return 0;
}
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git add .

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git commit -m "new file created in master branch"
[master 60d0c68] new file created in master branch
 1 file changed, 7 insertions(+)
 create mode 100644 code.cpp

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git remote
origin

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git push -u origin master
Enumerating objects: 4, done.
```

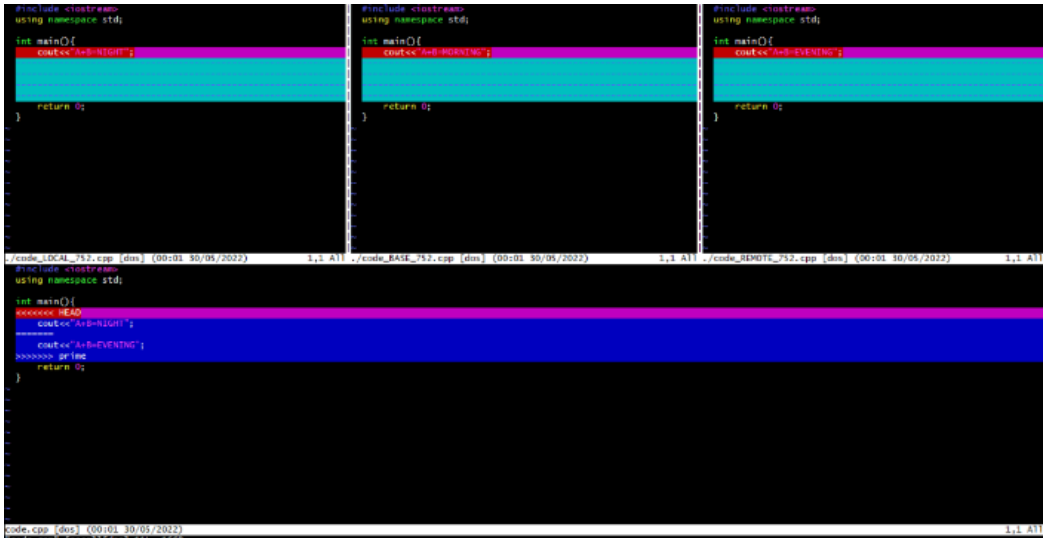
```
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git branch prime

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git checkout prime
Switched to branch 'prime'

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (prime)
$ git status
On branch prime
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   code.cpp

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

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (prime)
$ git add .

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (prime)
$ git commit -m "changes made in the prime branch"
[prime e55a24f] changes made in the prime branch
 1 file changed, 1 insertion(+), 1 deletion(-)

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (prime)
$ cat code.cpp
#include <iostream>
using namespace std;

int main(){
    cout<<"A+B=EVENING";
    return 0;
}
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (prime)
$ git push -u origin prime
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
```

```
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (prime)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   code.cpp

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

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git add .

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git commit -m "new change made"
[master 7d1ddd6] new change made
 1 file changed, 1 insertion(+), 1 deletion(-)

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ cat code.cpp
#include <iostream>
using namespace std;

int main(){
    cout<<"A+B=NIGHT";
    return 0;
}
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git push -u origin master
Enumerating objects: 5, done.
```

After we input this command, all of the files will be processed.

```
ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git merge prime
Auto-merging code.cpp
CONFLICT (content): Merge conflict in code.cpp
Automatic merge failed; fix conflicts and then commit the result.

ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master|MERGING)
$ cat code.cpp
#include <iostream>
using namespace std;

int main(){
<<<<<<< HEAD
    cout<<"A+B=NIGHT";
=======
    cout<<"A+B=EVENING";
>>>>>>> prime
    return 0;
}
ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master|MERGING)
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc codecompare smerge emerge vimdiff nvimdiff
Merging:
code.cpp

Normal merge conflict for 'code.cpp':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (vimdiff):
4 files to edit
```

After we input git mergetool command, all of the files will be processed and a dialog box will appear for the user to make the final changes in the file.

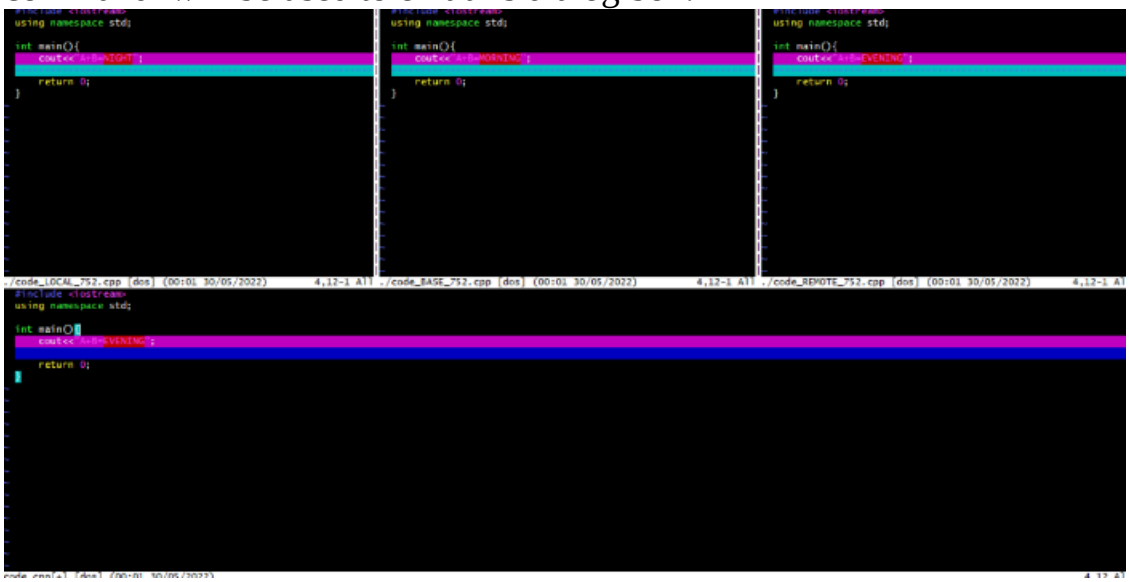


These are all the processes and the modifications done in the file. You can see three different files there, and you can see everything that was added or removed. After scrolling, you can verify where exactly the conflict happened. Manual modifications have allowed us to resolve file conflicts. Save the file and close the final file

Now it indicates the conflicting regions with special characters. The character sequences are like this:

- <<<<<<<
- =======
- >>>>>>>

Everything between <<<<<<< and ======= are your local changes. These changes are not in the remote repository yet. All the lines between ======= and >>>>>>> are the changes from the remote repository or another branch. Now you need to look into these two sections and make a decision.
it is entirely up to you to decide what changes are relevant to the situation. After your changes, you need to make sure that none of the conflict-indicating characters exist (<<<<<<<, =======, >>>>>>>) in the file. Now these are the final changes done which we want in the file. **:wq** command will be used to exit this dialog box.



git mergetool creates ***.orig** backup files while resolving merges. These are safe to remove once a file has been merged and its **git mergetool** session has completed.

Git add- stage the changes

Git commit – commit the changes with a message

Git push – finally now push the changes to the remote repository That's all there is to it to resolve the merge conflict in this scenario.

COLLABORTAOR'S ACTIVITY

- Step 0: Owner and collaborator ensure all changes are updated
- Step 1: Owner makes a change and commits
- Step 2: Collaborator makes a change and commits **on the same line**
- Step 3: Collaborator pushes the file to GitHub
- Step 4: Owner pushes their changes and gets an error
- Step 5: Owner pulls from GitHub to get Collaborator changes
- Step 6: Owner edits the file to resolve the conflict
- Step 7: Owner commits the resolved changes
- Step 8: Owner pushes the resolved changes to GitHub
- Step 9: Collaborator pulls the resolved changes from GitHub
- Step 10: Both can view commit history

Workflows to avoid merge conflicts

Some basic rules of thumb can avoid the vast majority of merge conflicts, saving a lot of time and frustration. These are words our teams live by:

- Communicate often
- Tell each other what you are working on
- Pull immediately before you commit or push
- Commit often in small chunks.

A good workflow is encapsulated as follows:

Pull -> Edit -> Add -> Pull -> Commit -> Push

Always start your working sessions with a pull to get any outstanding changes, then start doing your editing and work. Stage your changes, but before you commit, Pull again to see if any new changes have arrived. If so, they should merge in easily if you are working in different parts of the program. You can then Commit and immediately Push your changes safely. Good luck, and try to not get frustrated. Once you figure out how to handle merge conflicts, they can be avoided or dispatched when they occur, but it does take a bit of practice.

Experiment No. 09

Aim: RESET and REVERT

While Working with Git in certain situations we want to undo changes in the working area or index area, sometimes remove commits locally or remotely and we need to reverse those changes. There are 3 different ways in which we can undo the changes in our repository, these are **git checkout**, **git reset,** and **git revert**. Git checkout and git reset in fact can be used to manipulate commits or individual files. These commands can be confusing so it's important to find out the difference between them and to know which command should be used at a particular point of time.

Let us create a file **sample.txt** and "**Python** " written inside it.

We can see that we have done a single commit and the text document that has been committed with “Python” in it. These updates are currently in the working area and to see them we will see those using **git status**.

Now we have a change “**java**” which is untracked in our working repository and we need to discard this change. So, the command that we should use here is

Git checkout

git checkout is used to discard the changes in the working repository. git checkout <filename>

When we write git checkout command and see the status of our git repository and also the text document we can see that our changes are being discarded from the working directory and we are again back to the test document that we had before.

We want to unstage the file and the command that we would be using to unstage our file is –

Git reset

git reset is used when we want to unstage a file and bring our changes back to the working directory. git reset can also be used to remove commits from the local repository.

git reset HEAD <filename>

Whenever we unstage a file, all the changes are kept in the working area.

```
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git add sample.txt

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   sample.txt


Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git reset HEAD sample.txt
Unstaged changes after reset:
M       sample.txt

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   sample.txt

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

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$
```

We are back to the working directory, where our changes are present but the file is now unstaged. Now there are also some commits that we don't want to get committed and we want to remove them from our local repository. To see how to remove the commit from our local repository let's stage and commit the changes that we just did and then remove that commit.

We have 2 commits now, with the latest being the Added "another programming lang" commit which we are going to remove. The command that we would be using now is -

Git reset HEAD~1: Points to be noted -

HEAD~1 here means that we are going to remove the topmost commit or the latest commit that we have done.

We cannot remove a specific commit with the help of git reset, for example: we cannot say that we want to remove the second commit or the third commit, we can only remove latest commit or latest 2 commits ... latest N commits. (HEAD~n) [n here means n recent commits that needs to be deleted].

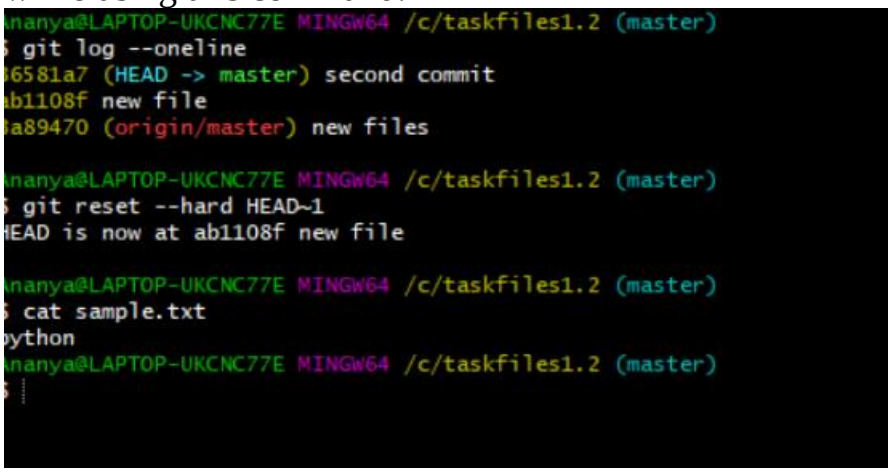
```
Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git add .

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git commit -m "third text added"
[master 42013b8] third text added
 1 file changed, 1 insertion(+)

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git reset HEAD~1
Unstaged changes after reset:
M       sample.txt

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   sample.txt

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

Ananya@LAPTOP-UKCNC77E MINGW64 /c/taskfiles1.2 (master)
```

After using the above command we can see that our commit is being deleted and also our file is again unstaged and is back to the working directory. There are different ways in which git reset can actually keep your changes.

git reset --soft HEAD~1 - This command will remove the commit but would not unstage a file. Our changes still would be in the staging area.

git reset --mixed HEAD~1 or **git reset HEAD~1 -** This is the default command that we have used in the above example which removes the commit as well as unstages the file and our changes are stored in the working directory.

git reset --hard HEAD~1 - This command removes the commit as well as the changes from your working directory. This command can also be called destructive command as we would not be able to get back the changes so be careful while using this command.



Points to keep in mind while using git reset command -

If our commits are not published to remote repository, then we can use git reset.

Use git reset only for removing commits that are present in our local directory and not in remote directory.

We cannot remove a specific commit with the help of git reset, for eg: we cannot say that we want to remove the second commit or the third commit, we can only remove latest commit or latest 2 commits ... latest N commits. (HEAD~n) [n here means n recent commits that needs to be deleted].

We just discussed above that the git reset command cannot be used to delete commits from the remote repository, then how do we remove the unwanted commits from the remote repository The command that we use here is –

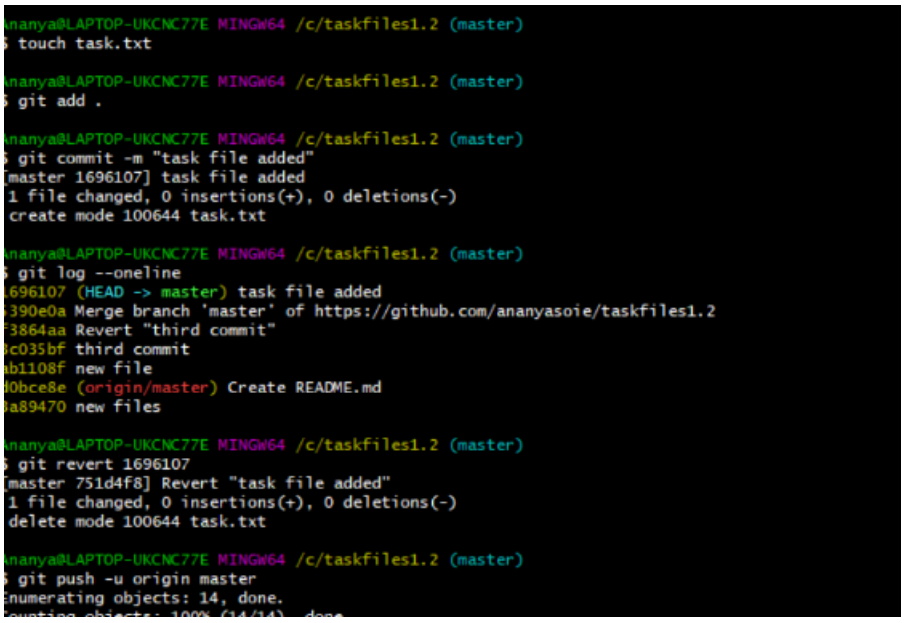
Git revert

git revert is used to remove the commits from the remote repository. Since now our changes are in the working directory, let's add those changes to the staging area and commit them.

Now let's push our changes to the remote repository.

Now we want to delete the commit that we just added to the remote repository. We could have used the git reset command but that would have deleted the commit just from the local repository and not the remote repository. If we do this then we would get conflict that the remote commit is not present locally. So, we do not use git reset here. The best we can use here is git revert.

git revert <commit id of the commit that needs to be removed>



Points to keep in mind –

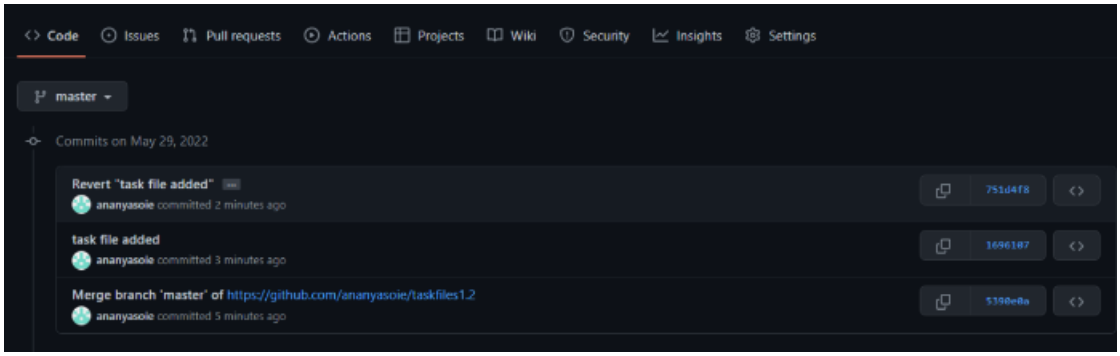
- Using git revert we can undo any commit, not like git reset where we could just remove "n" recent commits.

Now let's first understand what git revert does, git revert removes the commit that we have done but adds one more commit which tells us that the revert has been done. Let's look at the example –

Note: If you see the lines as we got after the git revert command, just visit your default text editor for git and commit the message from there, or it could directly take to you your default editor. We want a message here because when using git revert, it does not delete the commit instead makes a new commit that contains the removed changes from the commit.

We can see that the new commit is being added. However since this commit is in local repository so we need to do ***git push*** so that our remote repository also notices that the change has been done.

And as we can see we have a new commit in our remote repository and ***C++*** which we added in our 2nd commit is being removed from the local as well as the remote repository.



Let's summarize the points that we saw above –

Difference Table

git checkout	git reset	git revert
Discards the changes in the working repository.	Unstages a file and bring our changes back to the working directory	Removes the commits from the remote repository.
Used in the local repository.	Used in local repository	Used in the remote repository
Does not make any changes to the commit history.	Alters the existing commit history	Adds a new commit to the existing commit history.
Moves HEAD pointer to a specific commit.	Discards the uncommitted changes.	Rollbacks the changes which we have committed.
Can be used to manipulate commits or files.	Can be used to manipulate commits or files.	Does not manipulate your commits or files.

Project Work

Experiment No. 10

Project with Team Members - Demonstrating all aspects of Git

According to the given Task, each member has created a distributed repository and added team members, several task was done accordingly.

Opened and closed a pull request, each member created a pull request on other team member's repository and closed the pull request generated by other team members on the respective repository as being a maintainer. Later network graph was published.

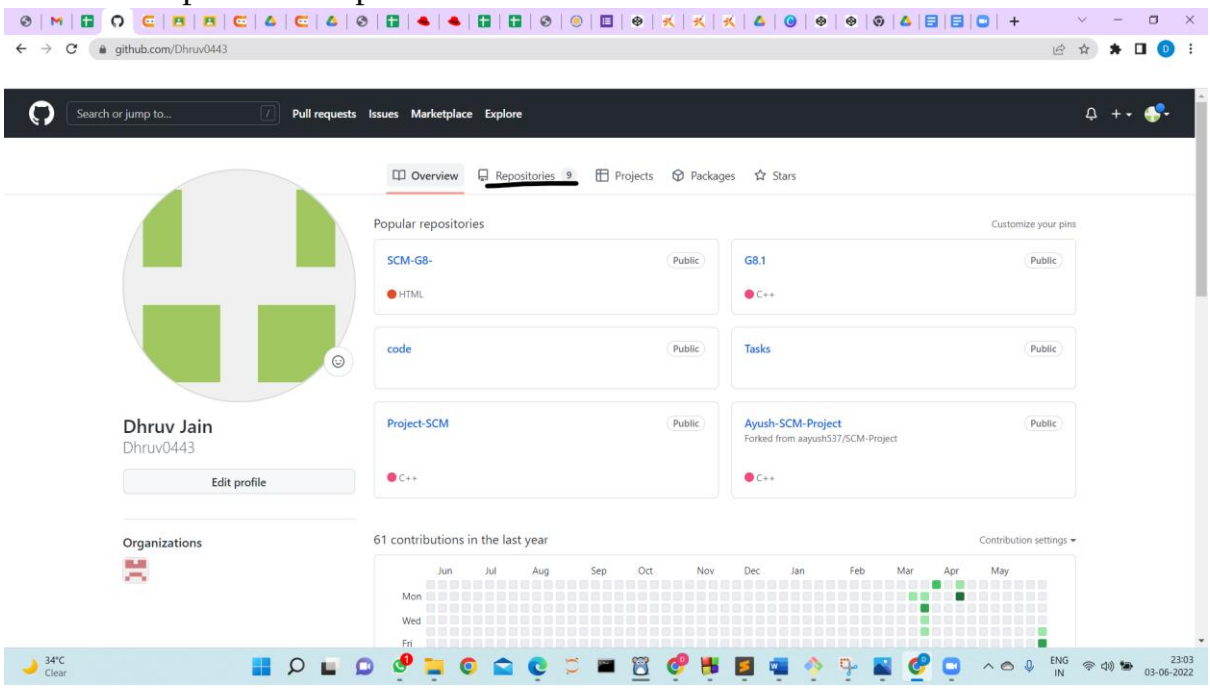
TEAM MEMBERS

- Aayush 2110990030
- Abhinn Singh Bisht 211099056
- Ankusha Sabharwal 2110990209
- Dhruv Jain 2110990443
- Gatik Veer 2110990493

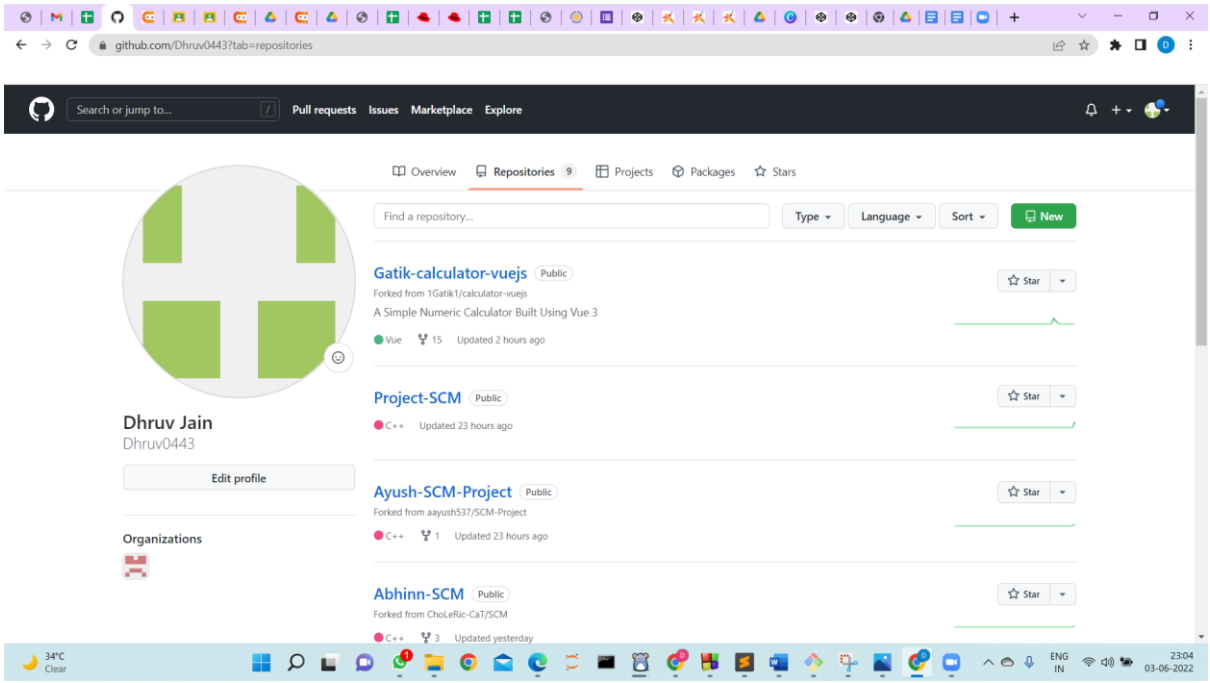
Aim: Create a distributed Repository and add member in Project team

CREATING REPOSITORY

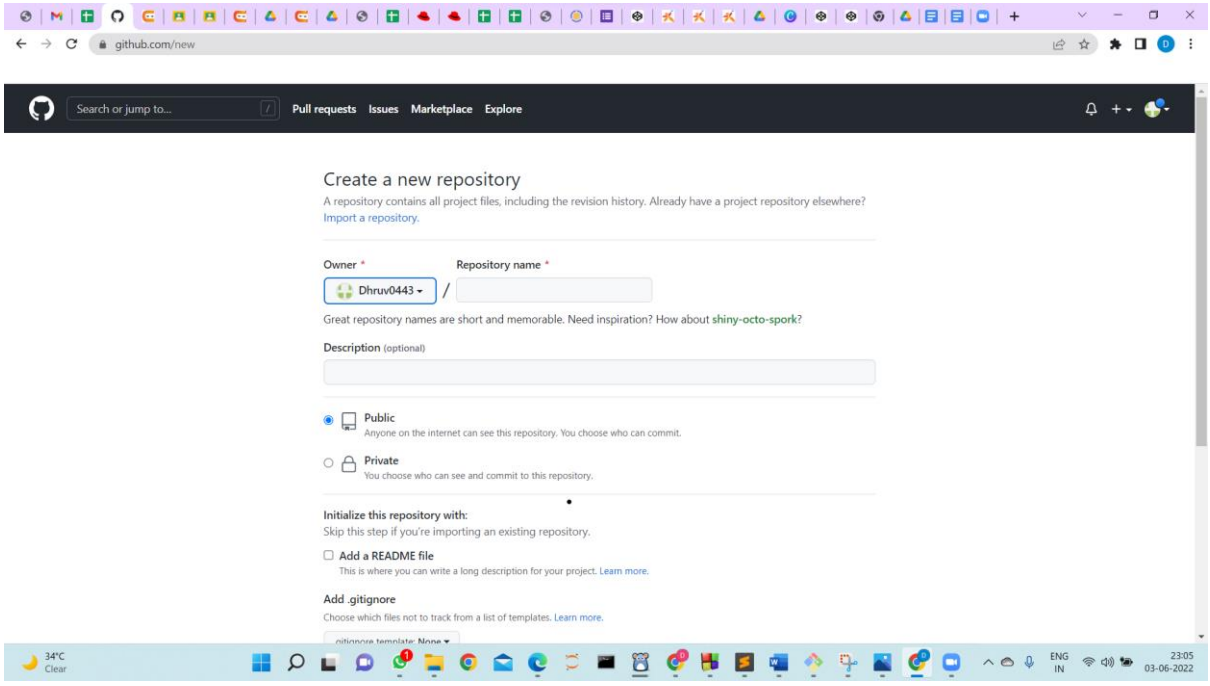
Click on Repositories option in the menu bar.



Click on the 'New' button on the top right corner.

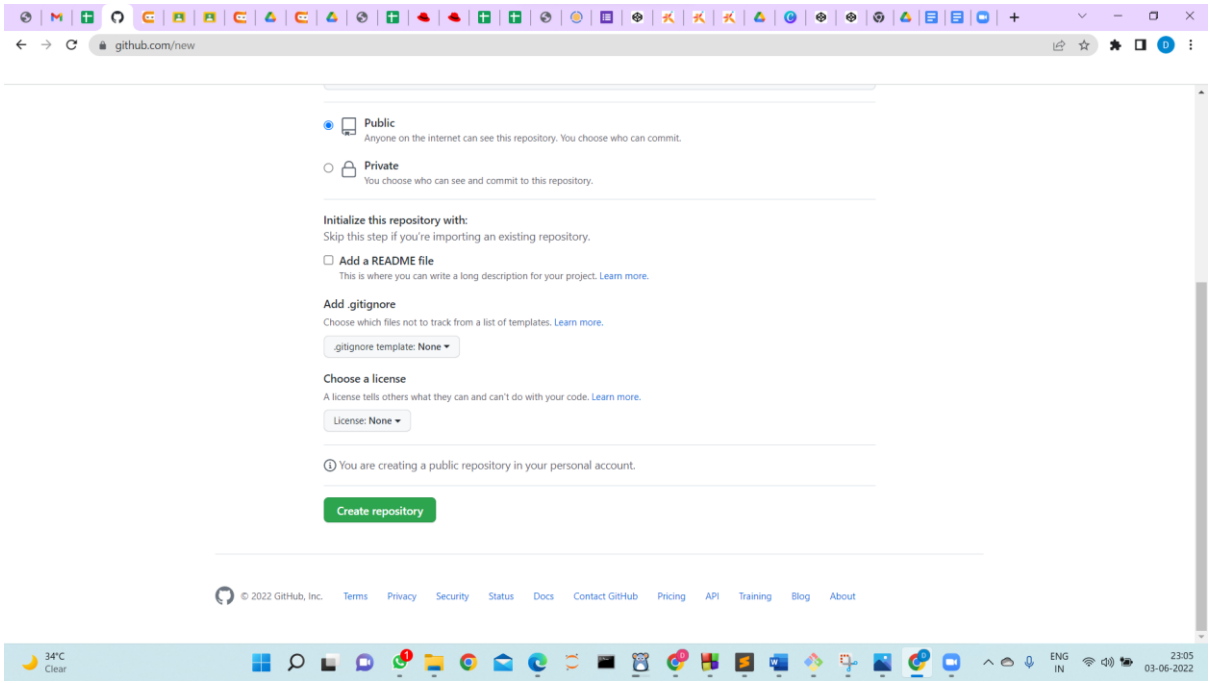


Enter the Repository name and add description if you want.



Select if you want the repository to be public or private.

Click on “Create Repository”.



Now, you have created your repository successfully.

ADDING COLLABORATORS

Step 1: Get the usernames of the GitHub users you will be adding as collaborators. In case, they are not on GitHub, ask them to sign in to GitHub.

Step 2: Go to your repository (intended to add collaborators)

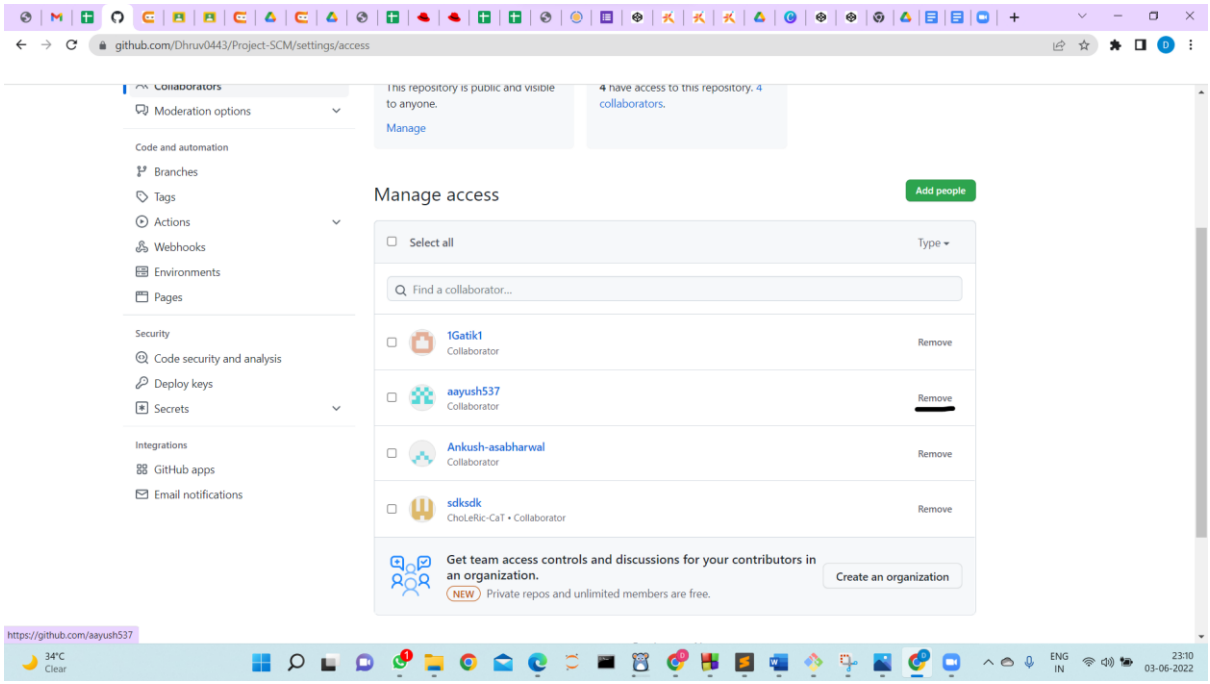
Step 3: Click into the Settings.

Step 4: A settings page will appear. Here, into the left-sidebar click into the Collaborators.
Step 5: Then a confirm password page may appear, enter your password for the confirmation.
Step 6: Next, click on Add People.
Step 7: Then a search field will appear, where you can enter the username of the ones you want to add as collaborator.
Step 8: After selecting the people, add them as collaborator.
Step 9: After sending the request for collaboration to that person whom we wanted as our collaborator for our project will get an email from the Team leader (by GitHub). He/she has to open its Email to view the invitation.
Step 10: After Clicking the View invitation, He/she will be redirected to the GitHub Page for accepting the invitation sent by the team leader.

Step 11: We are done adding a single collaborator. Now, they will get a mail regarding invitation to your repository. Once they accept, they will have collaborator access to your repository. Till then it will be in pending invitation state. You can also add more collaborator and delete the existing one as depicted below.

REMOVING COLLABORATOR PERMISSIONS FROM A PERSON CONTRIBUTING TO A REPOSITORY

Similar to the above steps, go to Your **Repository -> Settings -> Manage Access -> Remove** (on the right side of collaborator username)

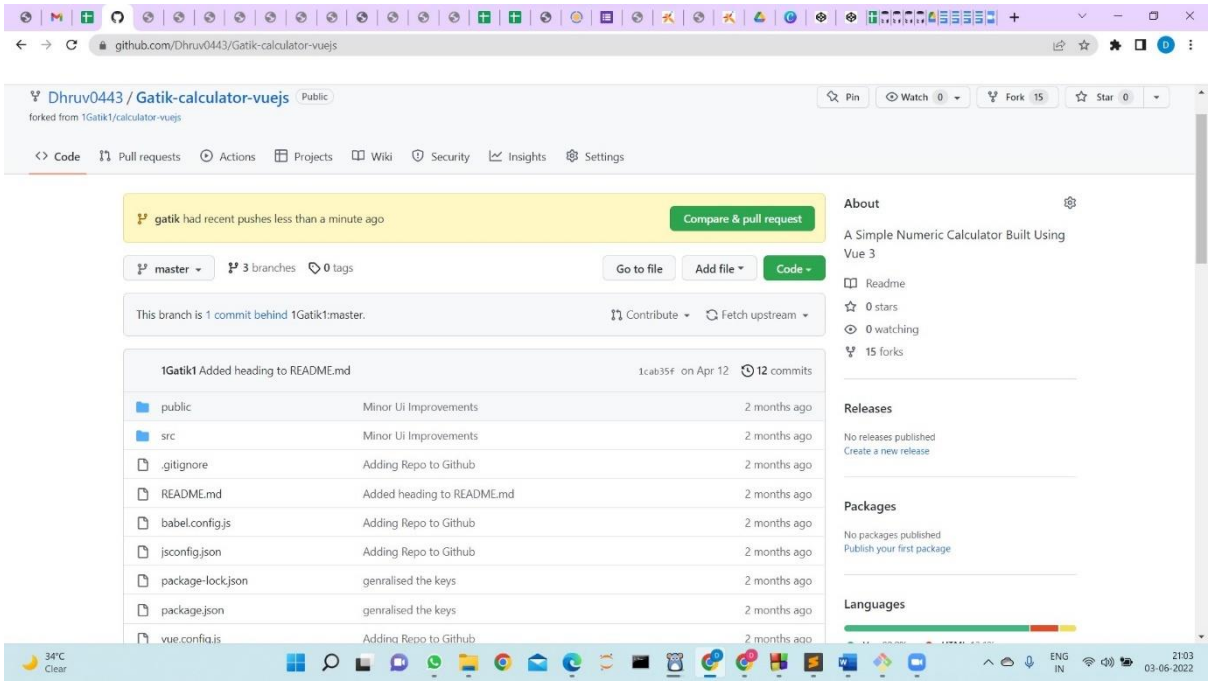


Aim- Open and Close a Pull Request

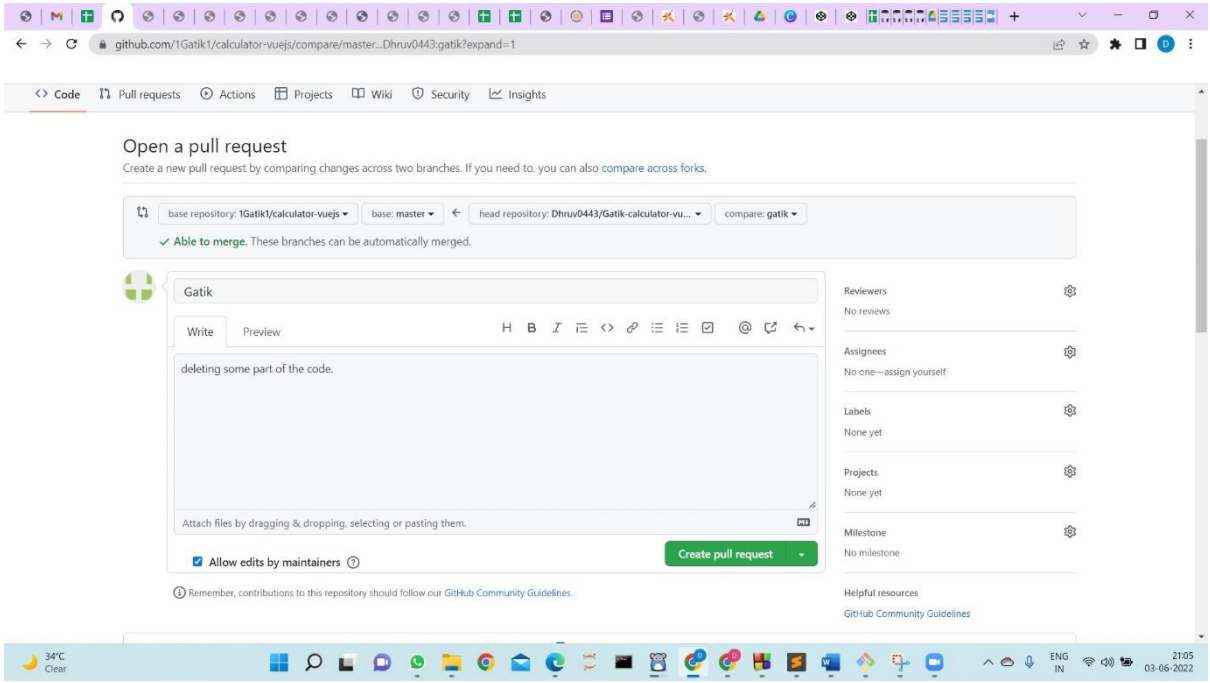
To open a pull request we first have to make a new branch, by using git branch **branchname** option.
After making new branch we add a file to the branch or make changes in the existing file.
Add and commit the changes to the local repository.
Use git push origin -u **branchname** option to push the new branch to the main repository.

```
DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik (main)
$ vi package.json

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik (main)
$ cd Gatik-calculator-vuejs

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ vi package.json

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git add package.json

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git commit -m'Deleting few lines'
[master 711a660] Deleting few lines
 1 file changed, 1 insertion(+), 5 deletions(-)

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git init
Reinitialized existing Git repository in D:/Gatik/Gatik-calculator-vuejs/.git/

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git remote add origin "https://github.com/Dhruv0443/Gatik-calculator-vuejs.git
"
error: remote origin already exists.

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git remote add origini "https://github.com/Dhruv0443/Gatik-calculator-vuejs.gi
t"

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git push -u origini master
fatal: unable to access 'https://github.com/Dhruv0443/Gatik-calculator-vuejs.git
/': Could not resolve host: github.com
```

After pushing new branch Github will either automatically ask you to create a pull request or you can create your own pull request.



Github will detect any conflicts and ask you to enter a description of your pull request.

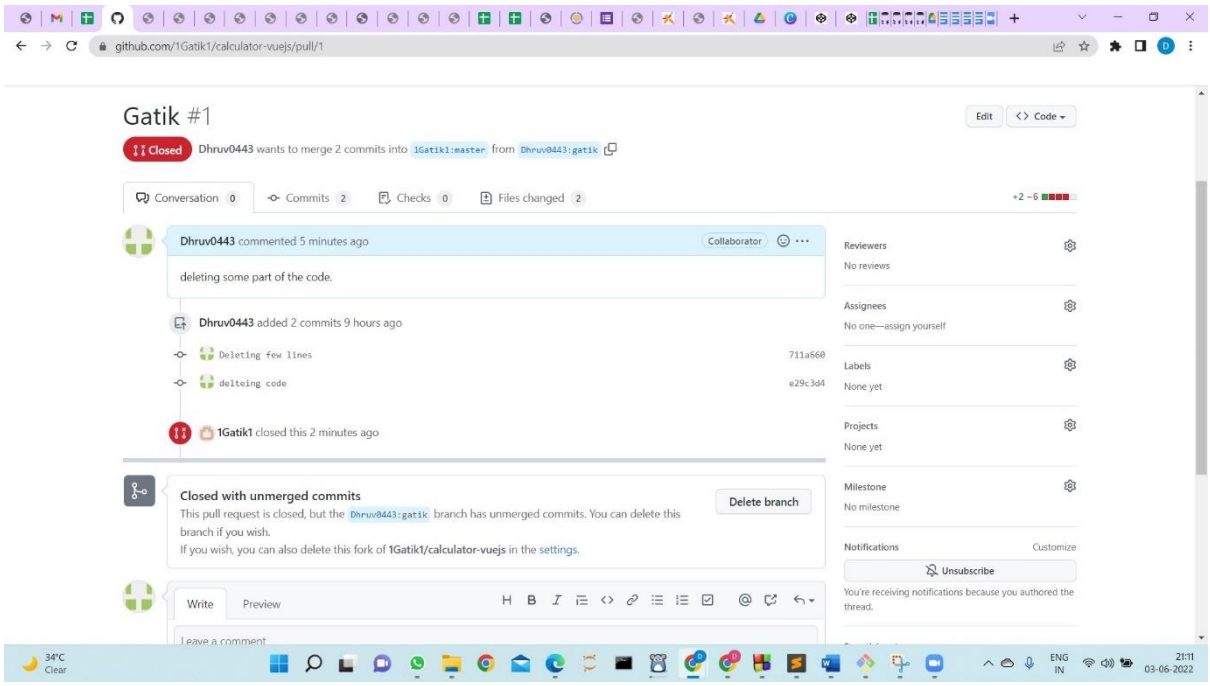


After opening a pull request,we have option to whether merge it or close the request.

If we chooses not to merge the pull request we will close the pull request.

To close the pull request we simply click on close pull request and add comment/ reason why we closed the pull request.

We can see all the pull requests generated and how they were dealt with by clicking on pull request option.



Aim - Create a Pull Request on a Team Member's Repository and close Pull Requests generated by Team Members on your own Repository as a Maintainer

CREATING A PULL REQUEST

As to create a pull request on a team member's repository, the first step is that we need to become a collaborator of his/her repository which we have that was discussed in the above task's.

Now as we have become a collaborator so in order to create a pull request on our team member's repository, we have to follow the steps given below:

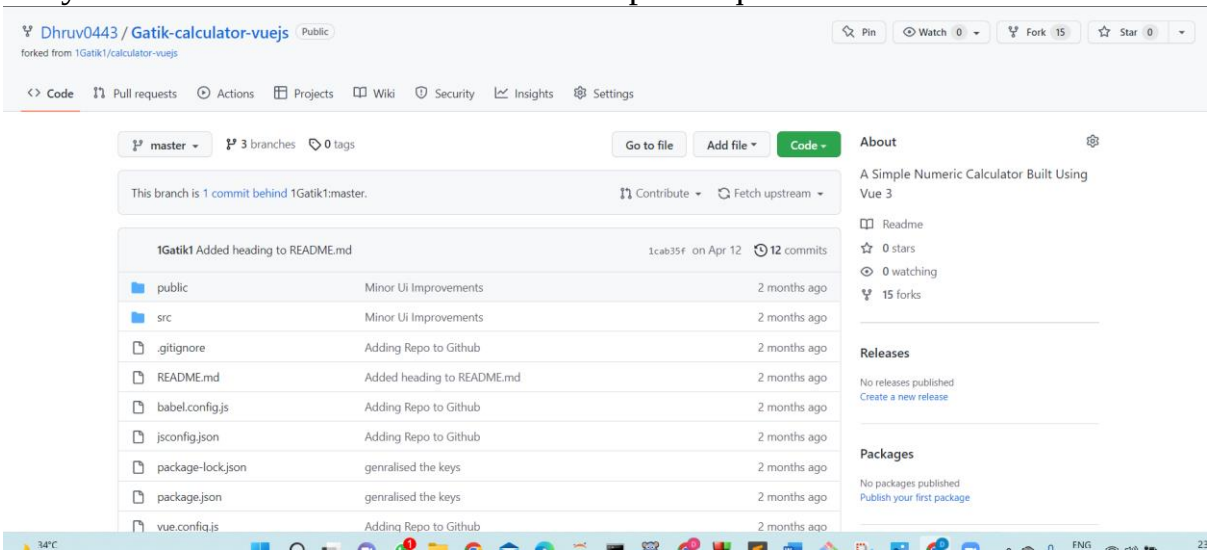
Create a new branch in the Remote Repository which we have forked from our team member's account (after becoming a collaborator of that particular repo).

```
DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik (master)
$ git branch -M main

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik (main)
$ git init
Reinitialized existing Git repository in D:/Gatik/.git/

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik (main)
$ git status
On branch main
```

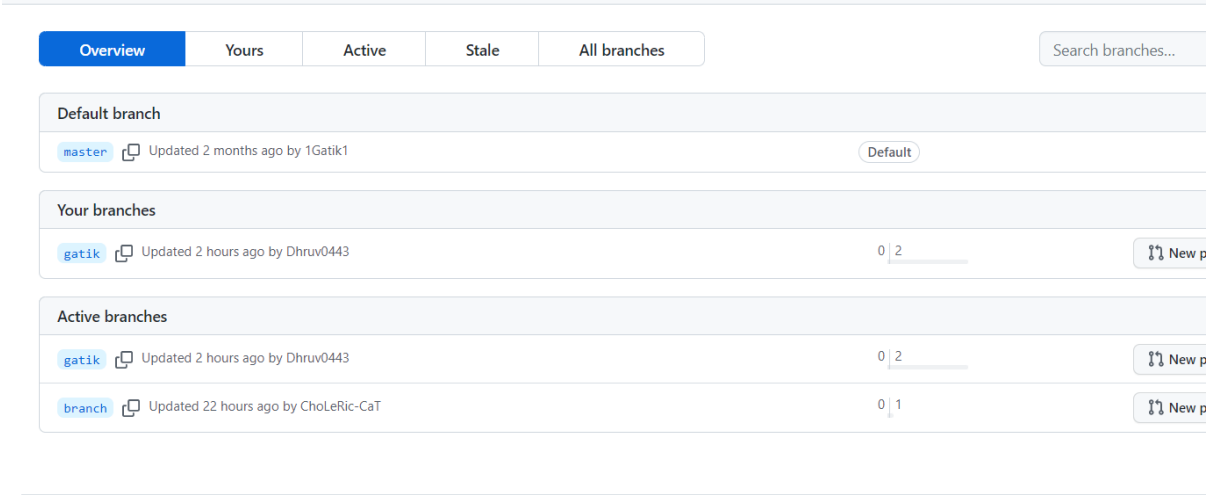
Now as we have created a new branch, we will add some files in this branch and push it on our team member's Repository on their Github account.

```
DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git add package.json

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git commit -m'Deleting few lines'
[master 711a660] Deleting few lines
 1 file changed, 1 insertion(+), 5 deletions(-)

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (master)
$ git init
Reinitialized existing Git repository in D:/Gatik/Gatik-calculator-vuejs/.git/

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (gatik)
$ git remote add gatik1 'https://github.com/Dhruv0443/Gatik-calculator-vuejs.git
'

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (gatik)
$ git push -u gatik1 gatik
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 583 bytes | 583.00 KiB/s, done.
Total 6 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 3 local objects.
remote:
remote: Create a pull request for 'gatik' on GitHub by visiting:
remote:      https://github.com/Dhruv0443/Gatik-calculator-vuejs/pull/new/gatik
remote:
To https://github.com/Dhruv0443/Gatik-calculator-vuejs.git
 * [new branch]      gatik -> gatik
branch 'gatik' set up to track 'gatik1/gatik'.

DELL@DESKTOP-VRD4NDB MINGW64 /d/Gatik/Gatik-calculator-vuejs (gatik)
$
```

Then we open a pull request by following the procedure from the above experiment.
The pull request is created and will be visible to our team member.
Now we ask him/her to login to their Github account.
They will notice a new notification in the pull request menu.



As they click on it. The pull request generated by us will be visible to them.

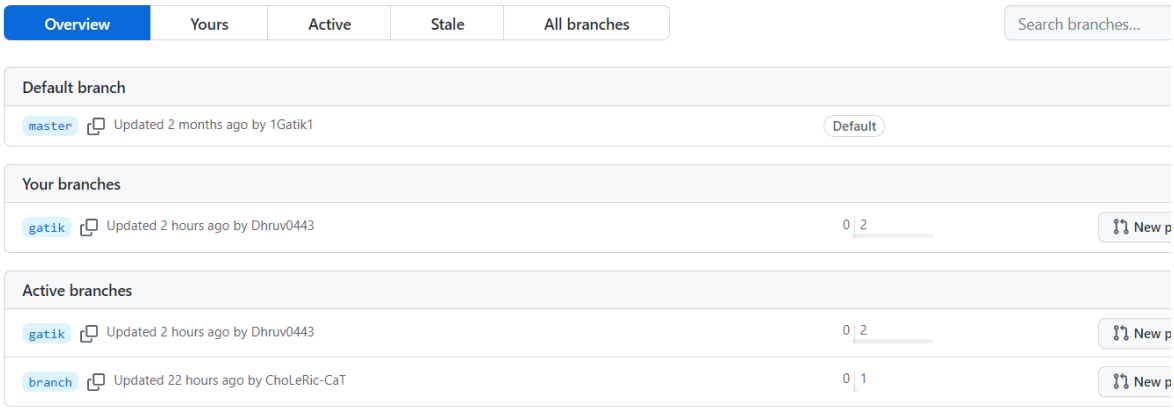


CLOSING THE PULL REQUESTS GENERATED BY OUR TEAM MEMBERS

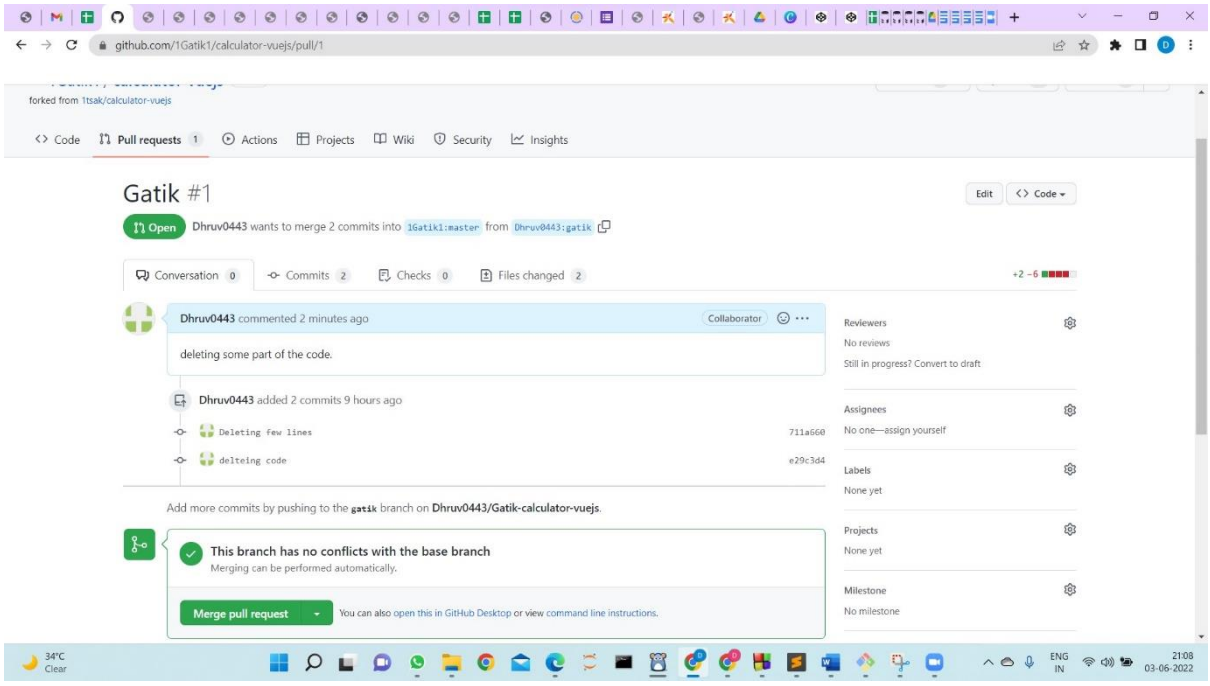
Now as we have also received some pull requests which have generated by our team members, we will learn how to close and merge these requests.

First we will learn how to close a pull request, so follow the steps given below:

Click on the pull request. Two options will be available, either to close the pull request or Merge the request in the main branch but for now we will close simply close the request.



By selecting the close pull request, the request will not accepted and not merged in the main branch.



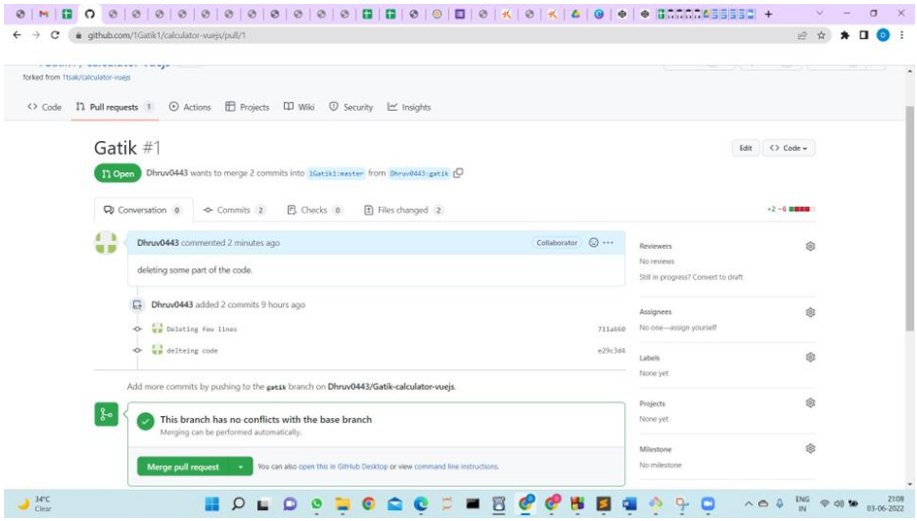
The closed pull request will appear at the “closed” option in Pull requests on the menu.

MERGING THE PULL REQUESTS GENERATED BY OUR TEAM MEMBERS

We have learned that how can we close a pull request, now it is time to see how can we merge these request in the main branch.

To merge these pull requests, follow the steps given below:

• The process is quite similar to closing the pull request but this time we will select the merge option.



• Thus, we conclude **opening** and **closing** of pull request. We also conclude merging of the pull request to the main branch.

Aim - Publish and Print Network Graphs

The network graph is one of the most useful feature for developers on GitHub. It is used to display the branch history of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.
A repository's graphs give you information on traffic, projects that depend on the repository, contributors and commits to the repository, and a repository's forks and network. If you maintain a repository, you can use this data to get a better understanding of who's using your repository and why they're using it.

Some repository graphs are available only in public repositories with GitHub

Free:

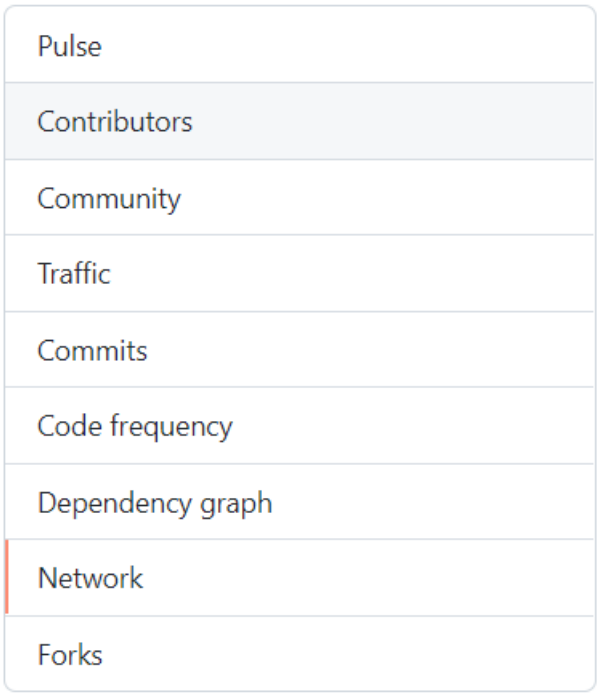
- Pulse
- Contributors
- Traffic
- Commits
- Code frequency
- Network

Steps to access network graphs of respective repository

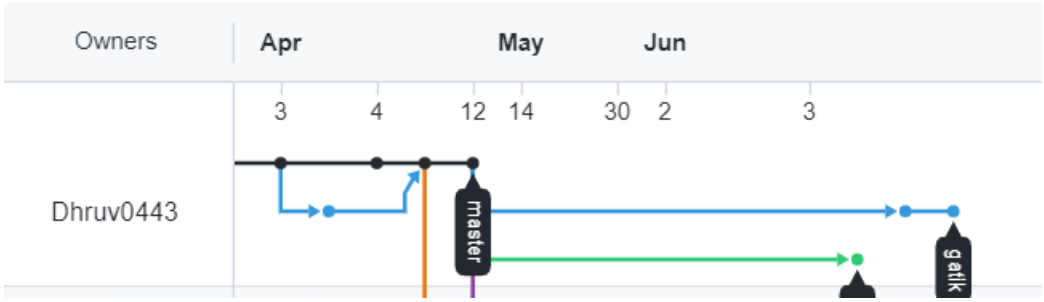
1. On GitHub.com, navigate to the main page of the repository.

2. Under your repository name, click **Insights**.

3.At the left sidebar, click on **Network**.



You will get the network graph of your repository which displays the branch history of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.



THANK YOU