Subject name: Source Code Management

Subject code: CS181

Cluster: Beta

Department: DCSE

Submitted by:
Gatik Veer
2110990493
G8-B

Submitted to:
Dr. Monit Kapoor

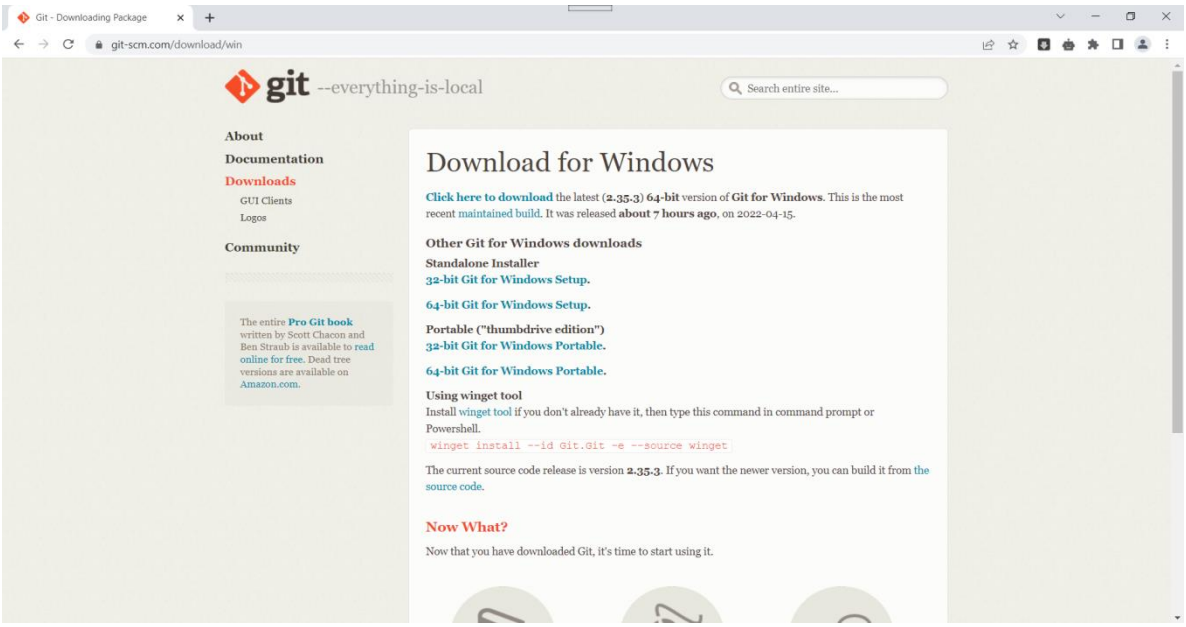
Table of Content -

Task 1.1

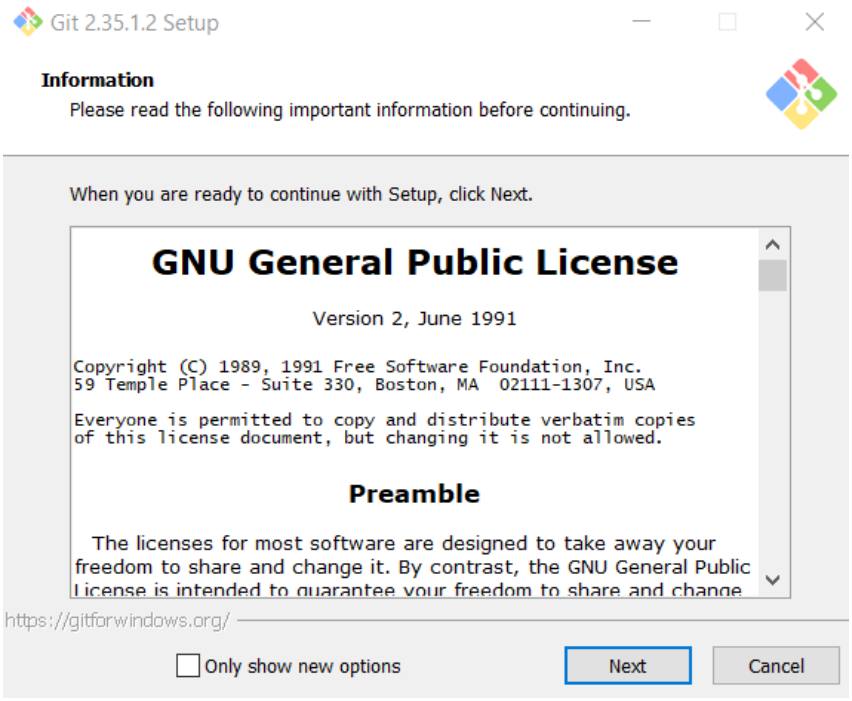
EXPERIMENT NO.1

AIM: Setting up Git Client

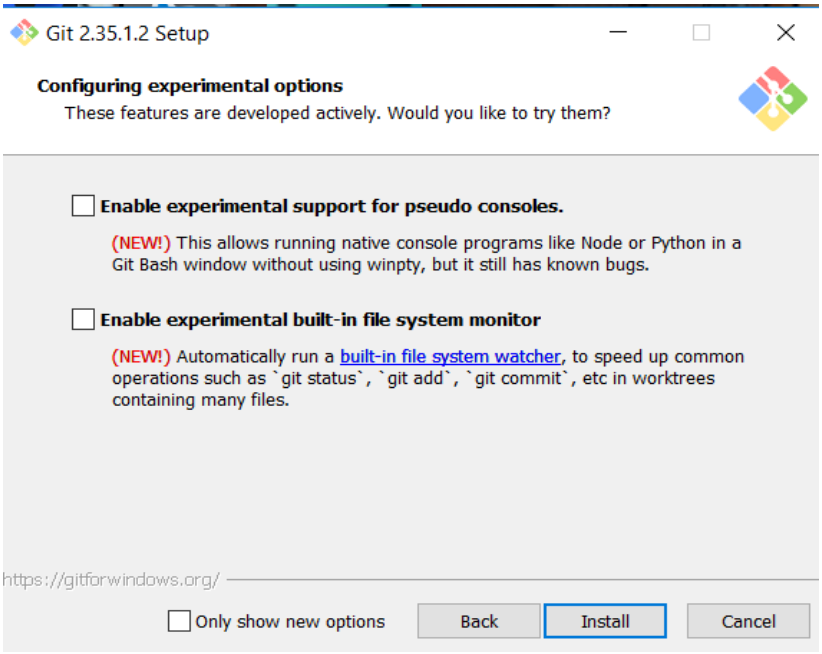
GIT Installation: Click the link below to download git client for windows -
https://git-scm.com/download/win



- Choose the operating system which suits your device . I'll choose the 64-bit Git for Windows setup.
- Then, open the Git in Download folder and the following window will appear. Click on Next.



❖ Continue clicking next without changing any of the default options selected until you reach the following window.



❖ Now click on install option.

After the installation of git we have to setup a GitHub account.

EXPERIMENT NO.2

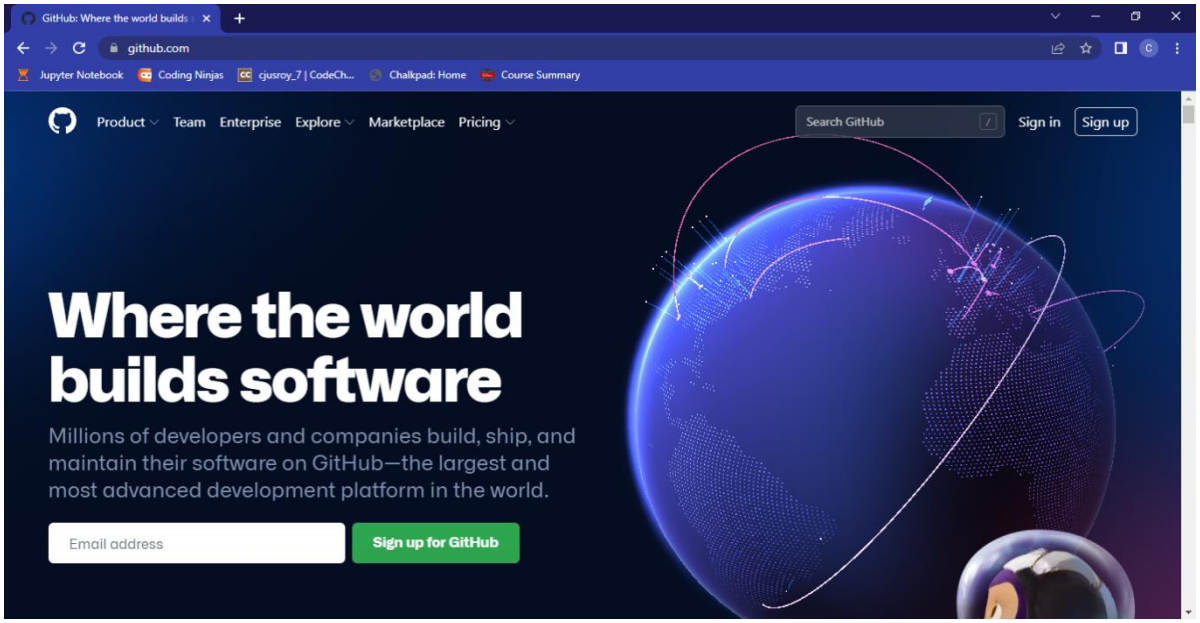
AIM: Setting up a GitHub Account.

First step in starting with GitHub are to create an account, choose a product that fits your needs best, verify your email, set up two-factor authentication, and view your profile.

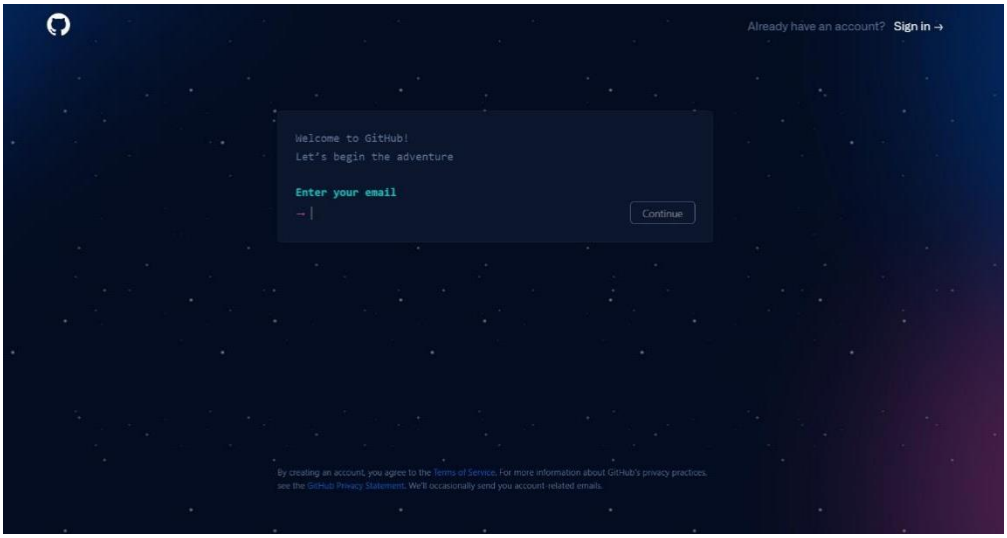
There are several types of accounts on GitHub. Every person who uses GitHub has their own user account, which can be part of multiple organisations and teams. Your user account is your identity on GitHub.com and represents you as an individual.

Creating an Account :

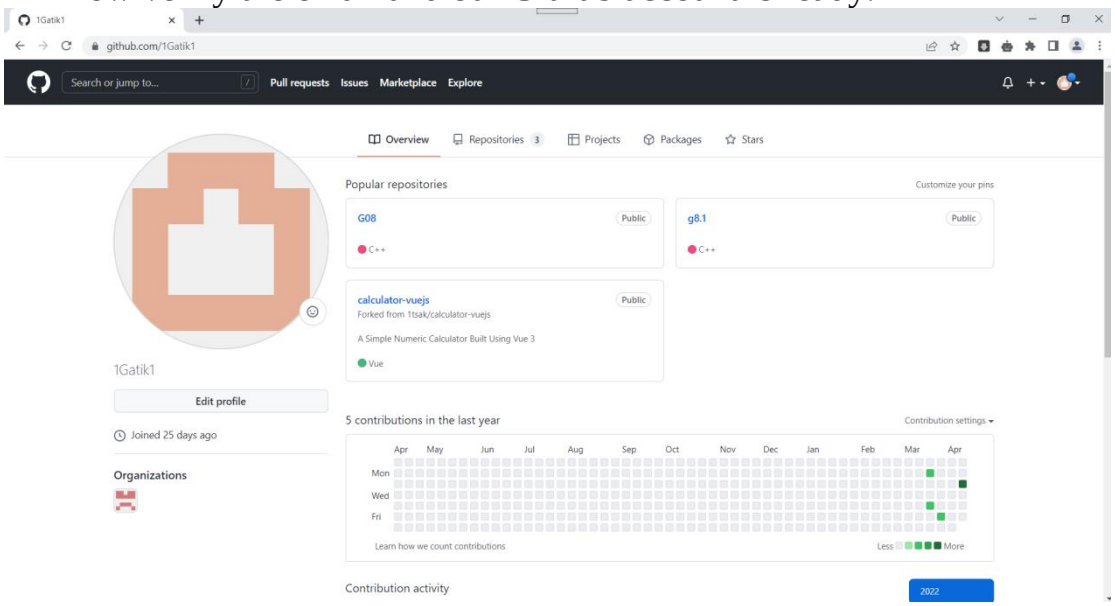
- Open your web browser and search http://github.com This will open the following page:



- Now click on ‘Sign Up’ on the upper-right conner of desktop window. This will open a new page where we must enter our email address for our account.



- Now type in the password and username you want to use for your GitHub account and click on 'Create Account'.
- Now verify the email and our Github account is ready.



EXPERIMENT NO.3

Aim: Generating Logs:

GIT BASICS:-

Firstly we need to form a connection between git bash and our github account before we can perform any other tasks on git.
To do this follow the following steps:

- Check your Git version using git version command

```
Alok Pasi@DESKTOP-D4EH9MS MINGW64 ~ (master)
$ git version
git version 2.35.1.windows.2
```

- Now let Git know where you are as this is important for version control systems,as each commit uses this info.You can specify Git configuration settings with the git config command.

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Now all your command will affect your github account.

Getting started with commands:

- ***git init*** - 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 hidden folder .git that contains all of your necessary repository files – a Git repository skeleton.

```
$ git init
```

- ***pwd (Print Working Directory)*** - It prints the path of the working directory, starting from the root.
- ***cd (Change Directory)*** - It allows one to change their directory.
- ***mkdir*** – The mkdir stands for '**Make Directory**'. With the help of mkdir command, you can create a new directory wherever you want in your system.
- ***ls*** - The **ls** is the list command. It will show the full list or content of your directory.

- ***ls -ah*** - The hidden files start with . (dot) symbol and they are not visible in the regular directory. The (ls -a) command will enlist the whole list of the current directory including the hidden files.

- ***git status*** - The git status command is used to display the state of the repository and staging area. It allows us to see the tracked, untracked files and changes. This command will not show any commit records or information.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

- ***git add*** - In review, git add is the first command in a chain of operations that directs Git to "save" a snapshot of the current project state, into the commit history. When used on its own, git add will promote pending changes from the working directory to the staging area.

```
git add [file]
```

add a file as it looks now to your next commit (stage)

- ***git commit*** - A commit, or "revision", is an individual change to a file (or set of files). It's like when you save a file, except with Git, every time you save it creates a unique ID (a.k.a. the "SHA" or "hash") that allows you to keep record of what changes were made when and by who.

```
git commit -m "[descriptive message]"
```

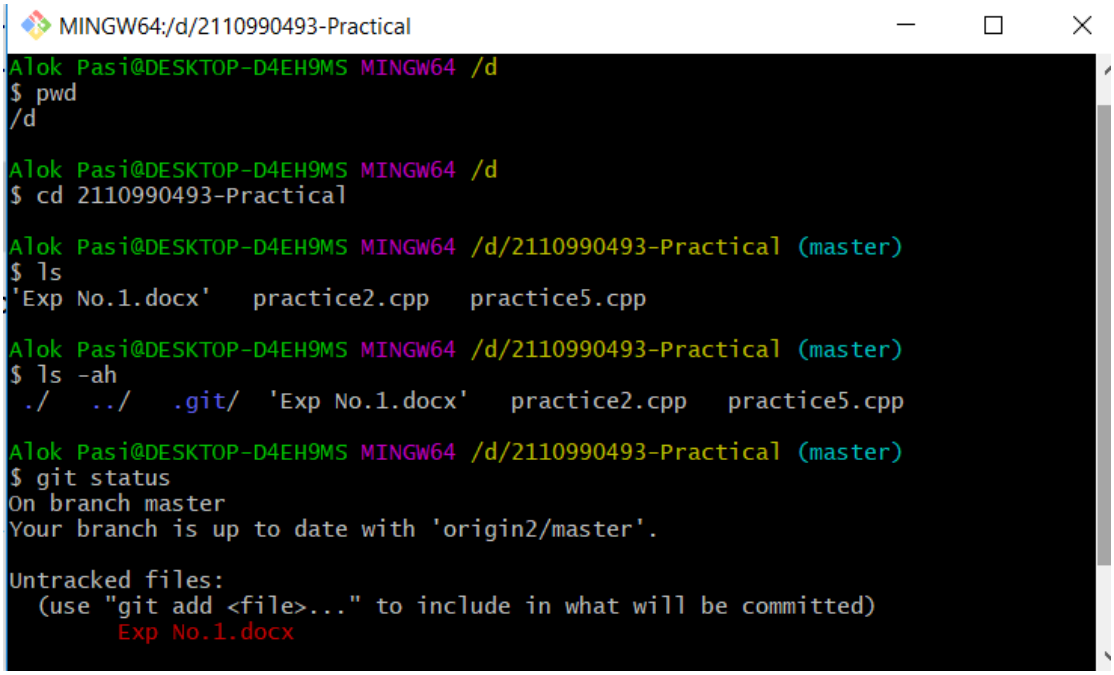
commit your staged content as a new commit snapshot

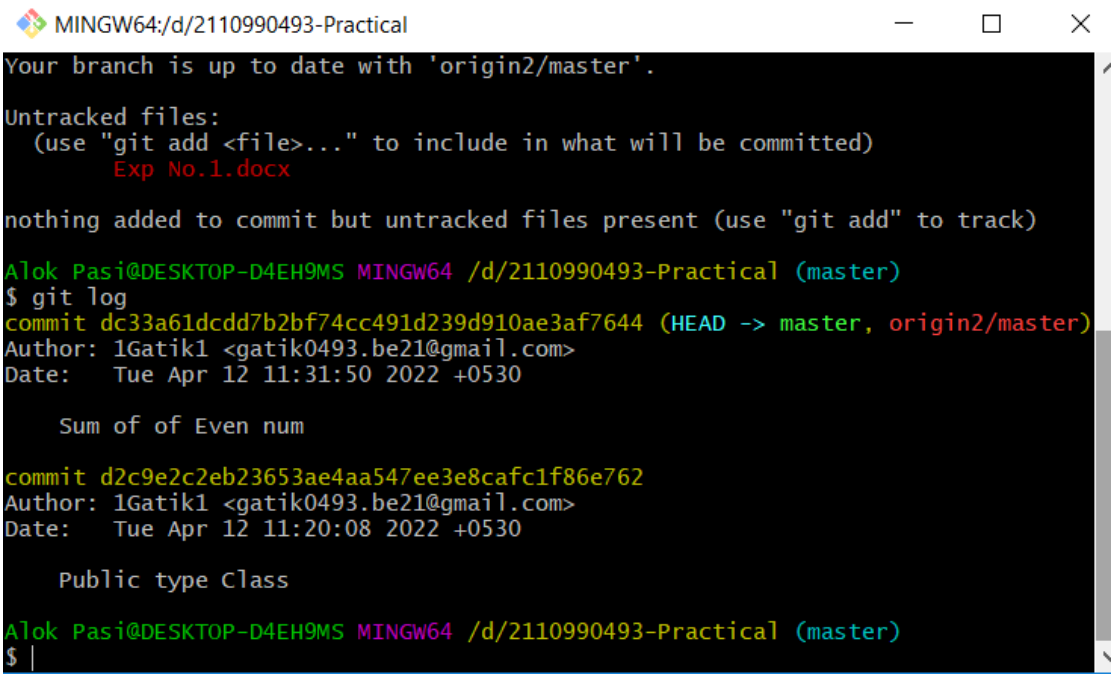
- ***git log*** - Git log command is one of the most usual commands of git. It is the most useful command for Git. Every time you need to check the history, you have to use the git log command. The basic git log command will display the most recent commits and the status of the head.

```
git log
```

show all commits in the current branch's history

Creating a new Log :





MINGW64:/d/2110990493-Practical

```
Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$ git add .

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$ git status
On branch master
Your branch is up to date with 'origin2/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   Exp No.1.docx


Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$ git commit -m "Setting up Git Client"
[master f54f430] Setting up Git Client
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Exp No.1.docx

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$
```

MINGW64:/d/2110990493-Practical

```
Your branch is up to date with 'origin2/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   Exp No.1.docx


Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$ git commit -m "Setting up Git Client"
[master f54f430] Setting up Git Client
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Exp No.1.docx

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$ git remote
origin2

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$ git push -u origin2 master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 625.99 KiB | 13.32 MiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Group08-Chitkara-University/2110990493.git
   dc33a61..f54f430  master -> master
branch 'master' set up to track 'origin2/master'.

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$ git status
On branch master
Your branch is up to date with 'origin2/master'.

nothing to commit, working tree clean

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$ git log
commit f54f430b81da5d8686bff203b3ecbfd7560206ad (HEAD -> master, origin2/master)
Author: 1Gatik1 <gatik0493.be21@gmail.com>
Date:   Fri Apr 15 18:28:00 2022 +0530

    Setting up Git Client

commit dc33a61dcdd7b2bf74cc491d239d910ae3af7644
Author: 1Gatik1 <gatik0493.be21@gmail.com>
Date:   Tue Apr 12 11:31:50 2022 +0530

    Sum of of Even num

commit d2c9e2c2eb23653ae4aa547ee3e8cafc1f86e762
Author: 1Gatik1 <gatik0493.be21@gmail.com>
Date:   Tue Apr 12 11:20:08 2022 +0530

    Public type Class

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/2110990493-Practical (master)
$
```

EXPERIMENT NO. 4

AIM: Create and Visualize Branches

How to create branches?

The main branch in git is called **master branch**. But we can make branches out of this main master branch. All the files present in master can be shown in branch but the files which are created in branch are not shown in master branch. We also can merge both the parent(master) and child (other branches).

1. For creating a new branch: **git branch "name of branch**"
2. To check how many branches we have - **git branch**
3. To change the present working branch: **git checkout "name of the branch**"

```
$ git branch testing
```

```
$ git checkout master
Switched to branch 'master'
```

Visualizing Branches :

To visualize, I have to edited a file in the branch "feature" instead of the master branch. After this I have done three step architecture i.e. working directory, staging area and git repository.

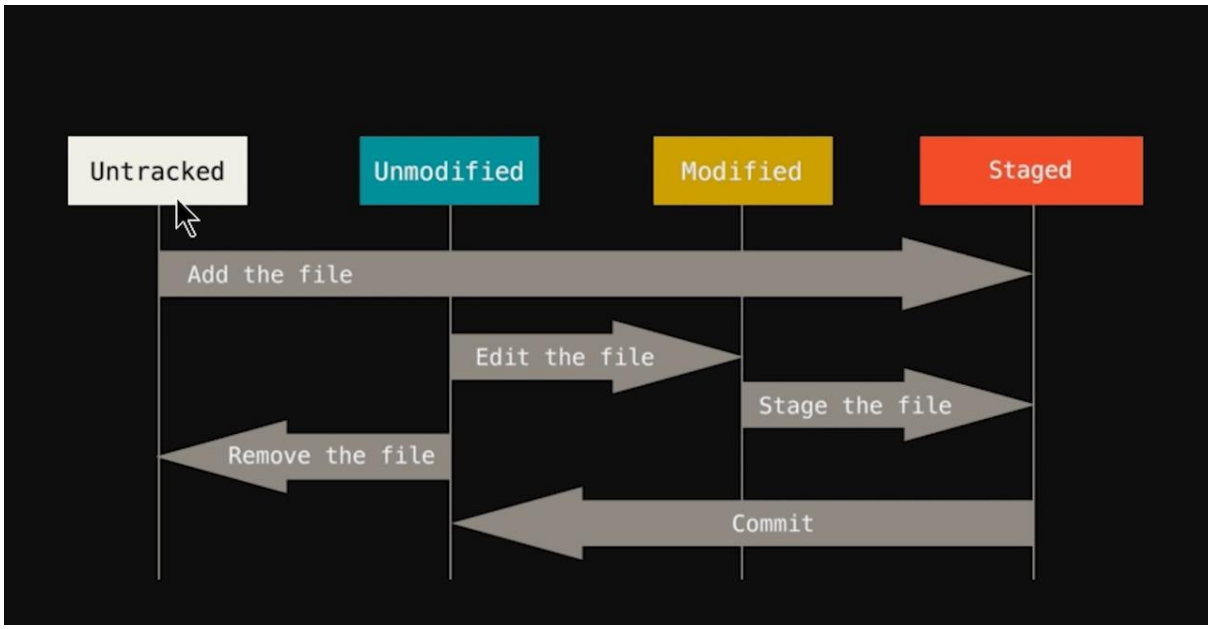
```
Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (master)
$ git branch
  checkout
  feature
* master

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (master)
$ git checkout feature
Switched to branch 'feature'
Your branch is ahead of 'origin/feature' by 1 commit.
  (use "git push" to publish your local commits)

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (feature)
$ ls
hello.cpp  img/

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (feature)
$ vi hello.cpp

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (feature)
$ git status
On branch feature
Your branch is ahead of 'origin/feature' 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:   hello.cpp

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

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (feature)
$ git add .

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (feature)
$ git commit -m "For Experiment"
[feature 0d897fc] For Experiment
 1 file changed, 1 insertion(+), 1 deletion(-)

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (feature)
$ git remote
origin
origin1

Alok Pasi@DESKTOP-D4EH9MS MINGW64 /d/g8 (feature)
$ git push -u origin feature
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 592 bytes | 296.00 KiB/s, done.
Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/1Gatik1/G08
   7d3b495..0d897fc  feature -> feature
branch 'feature' set up to track 'origin/feature'.
```

EXPERIMENT NO.5

AIM: Git life cycle

Git is used in our day-to-day work, we use Git for keeping a track of our files, working in a collaboration with our team, to go back to our previous code versions if we face some error. Git helps us in many ways. Let us look at the Lifecycle description that git has and understand more about its life cycle. Let us see some of the basic steps that we have to follow while working with Git :

- You clone the Git repository as a working copy.
- You modify the working copy by adding/editing files.
- If necessary, you also update the working copy by taking other developer's changes.
- You review the changes before commit.
- You commit changes. If everything is fine, then you push the changes to the repository.
- After committing, if you realize something is wrong, then you correct the last commit and push the changes to the repository.



➢ WORKING DIRECTORY

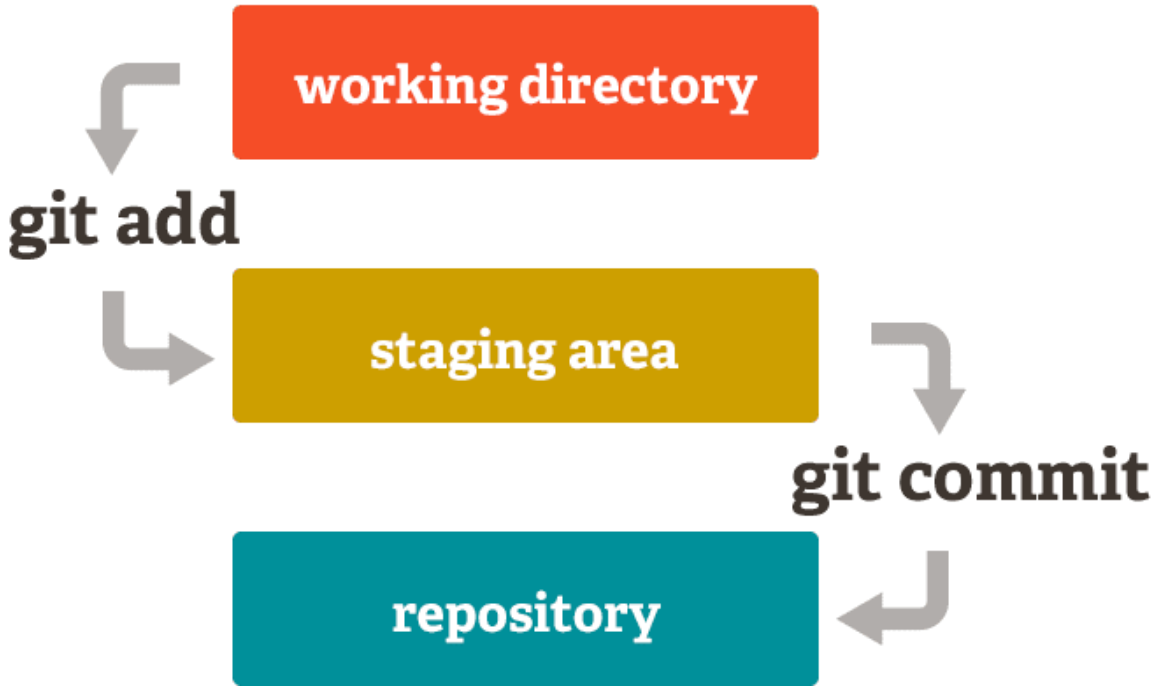
This is where you work with your files as you are updating them. This is on your local machine.

➢ STAGING AREA

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.

➢ GIT DIRECTORY

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.



Task 1.2

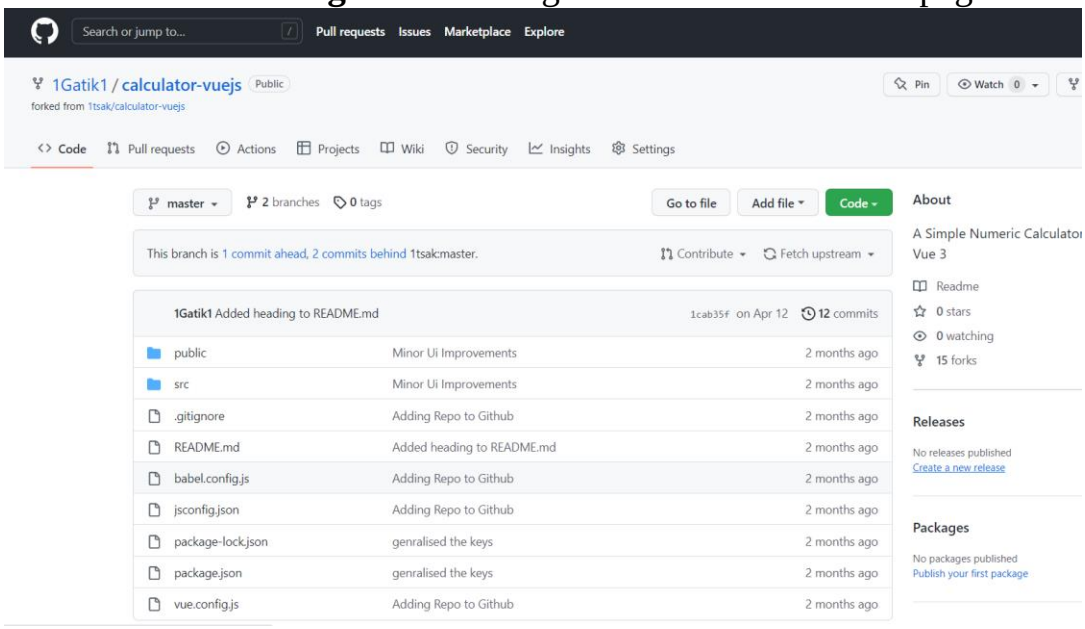
Practical No. 1

Aim: Add collaborators on GitHub Repository

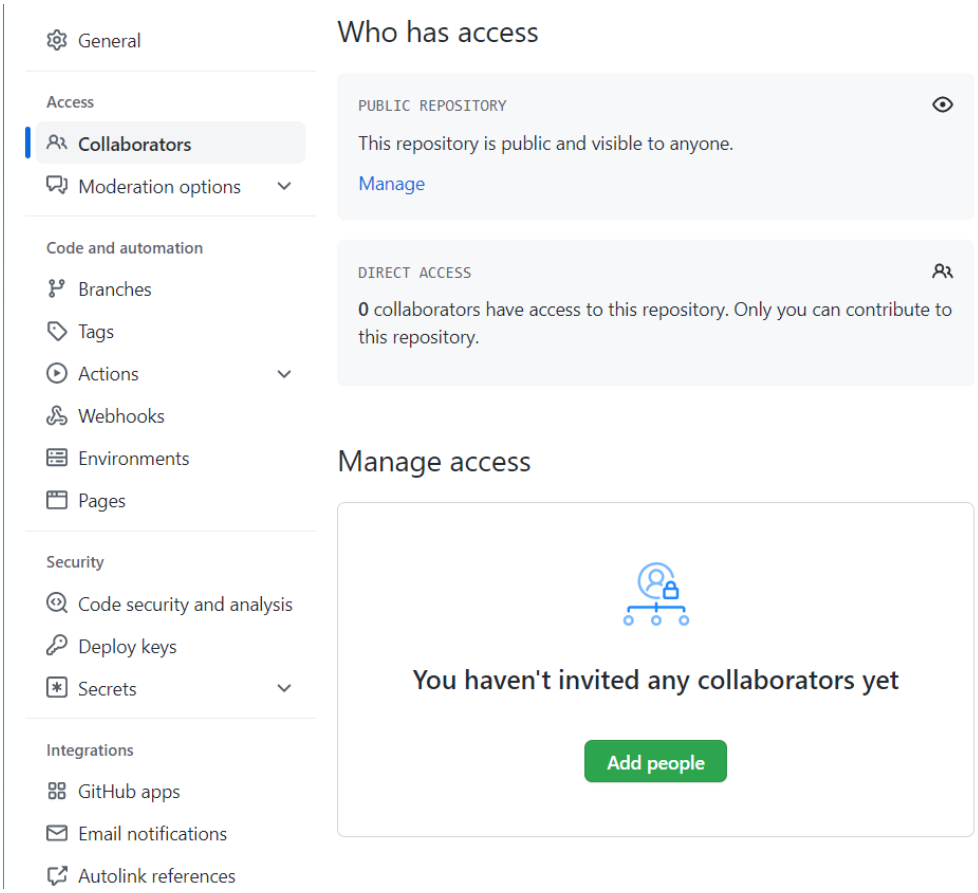
Even if you have a public repository in GitHub, not everyone has the permission to push code into your repository. Other users have a read-only access and cannot modify the repository. In order to allow other individuals to make changes to your repository, you need to invite them to collaborate to the project.

The following steps should be performed to invite other team members to collaborate with your repository.

1. Click on the **Settings** tab in the right corner of the GitHub page.

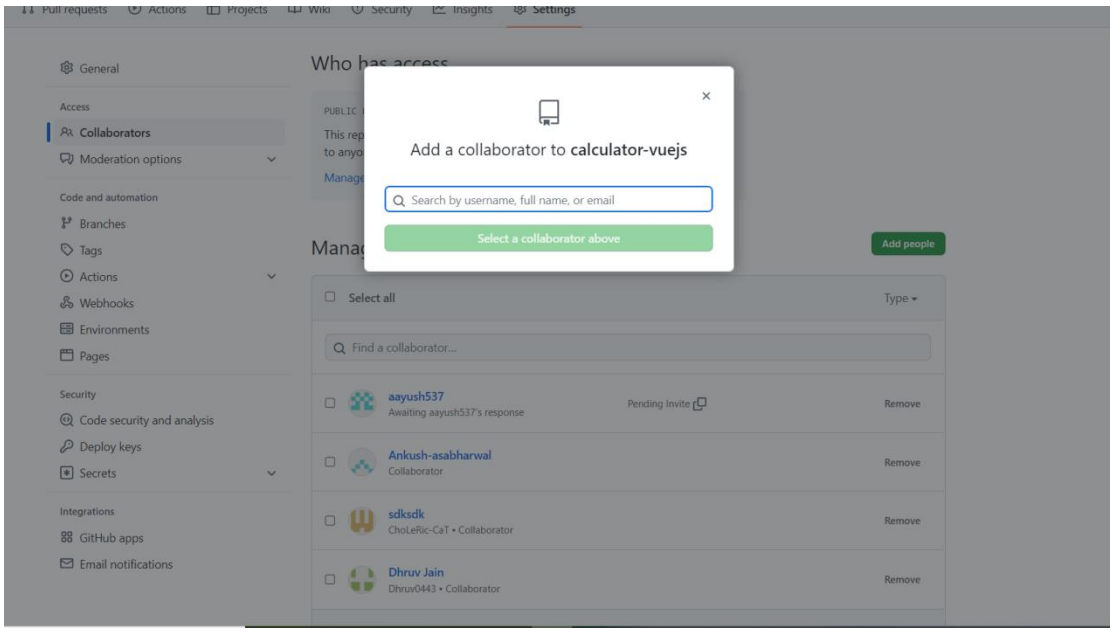


2. Go to **Collaborators** option under the Access tab. On the Manage Access page, you will see an **Invite collaborator link** as shown in the below diagram



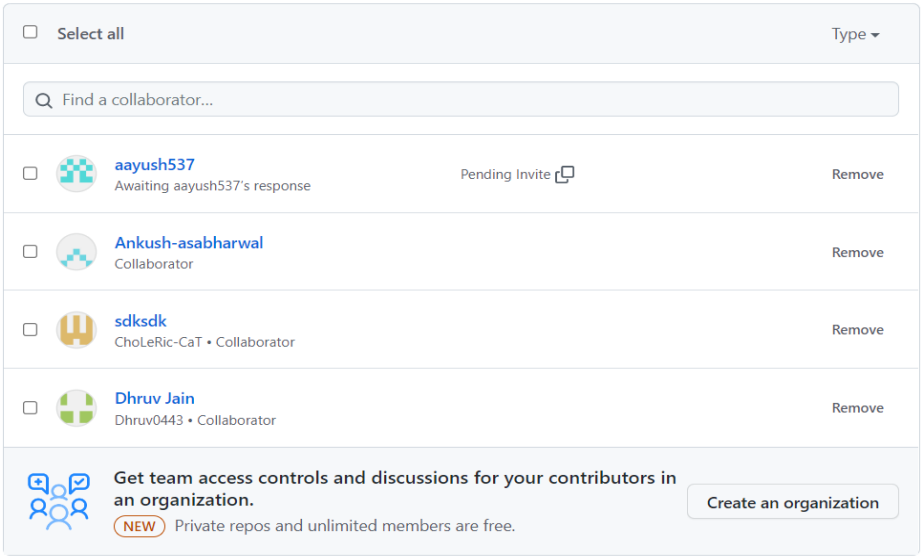
3. You can Invite collaborators by any of the following options −

- Username
- Full name
- Email



4. After you send the invite, the collaborator receives an email invitation. The collaborator has to accept it in order to get permission to collaborate on the same project

5. The **Manage Access** option also allows a repository owner to view the invitations that are
pending and not accepted.

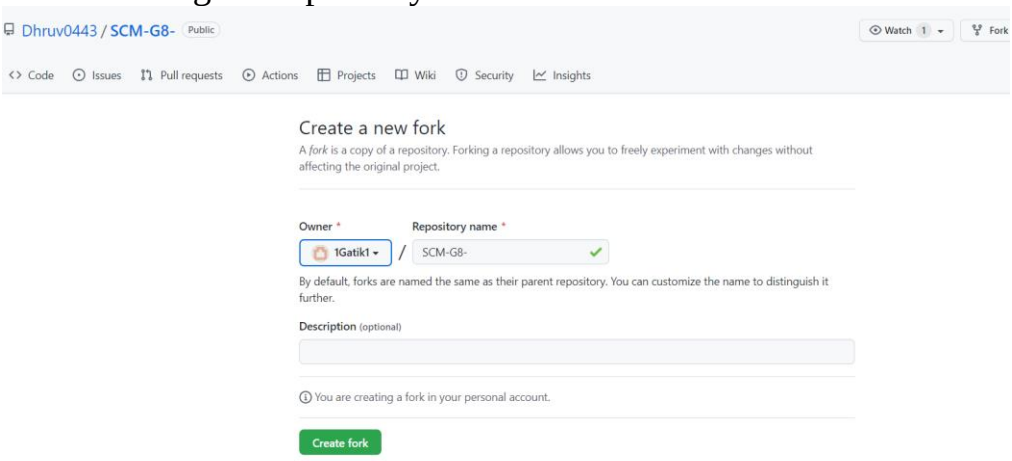


Practical No. 2

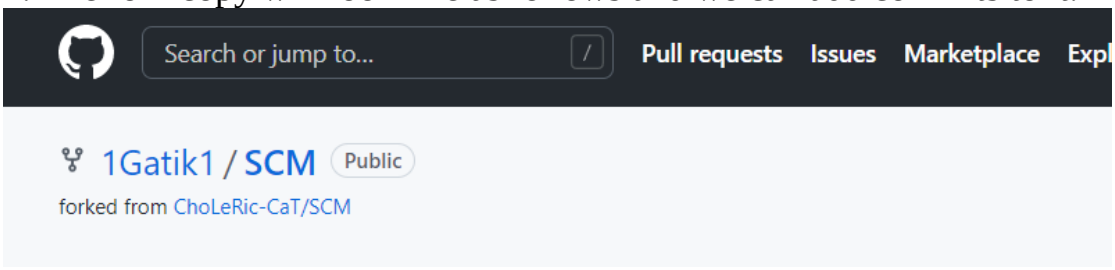
Aim: Fork And Commit

A **fork** is a copy of a repository. This is useful when you want to contribute to someone else's project or start your own project based on theirs. **fork** is not a command in Git, but something offered in GitHub and other repository hosts.

1. We can see the fork option at the top right corner of a repository page. By clicking on that, the forking process will start. It will take a while to make a copy of the project for other users. After the forking completed, a copy of the repository will be copied to your GitHub account. It will not affect the original repository.



2. The fork copy will look like as follows and we can add commits to it.



3. We can freely make changes and then create a pull request for the main project. The owner of the project will see your suggestion and decide whether they want to merge the changes or not.

Practical No. 3

Aim: Merge and resolve conflicts created due to own activity and collaborators activity.

Merging Branches

Once you've completed work on your branch, it is time to merge it into the main branch. Merging takes your branch changes and implements them into the main branch.

In real world, when we merge branches, we will run into conflicts quite often. Conflict happens because of the following reasons −

- When the same line of code is changed in different ways in two branches.
- A given file is changed in one branch but deleted in another branch.
- Same file is added twice in two different branches but the content of the file is different.

In these cases, git will stop the merge process as it cannot figure out how to merge the changes. In such scenarios we need to intervene manually and instruct how to proceed with the merging process.

Resolve conflicts created due to own activity and collaborator's activity:

1. Making some changes in the collaboration repository.

2. Collaborators have also made some changes in this repository on the other branch

3. We will now try to merge the changes from the 'branch' branch to the master branch.

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/calculator-vuejs/public
master)
$ git merge branch
Auto-merging public/index.html
CONFLICT (content): Merge conflict in public/index.html
Automatic merge failed; fix conflicts and then commit the result.
```

The output shows that the branch is now in an intermediate state of merging as the automatic merging has failed due to the conflict. We would need to interfere manually to complete the merging process.

4. Check the contents of the file index.html

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/calculator-vuejs/p
ublic (master|MERGING)
$ cat index.html
<!DOCTYPE html>
<html lang="">
  <head>
<<<<<<< HEAD
=======

    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
>>>>>>> branch
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
<<<<<<< HEAD

      <strong>We are very sorry but <%= htmlWebpackPlugin.options.title %
> doesn't work properly without JavaScript enabled. Please enable it to c
ontinue.</strong>
=======
      <strong>We're not sorry but <%= htmlWebpackPlugin.options.title %>
doesn't work properly without JavaScript enabled. Please enable it to con
tinue.</strong>
>>>>>>> branch

    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

From the contents of the file, it is clear that the different parts of the code will be considered from both the branches. The output screen shows a divider with the "=======" notation. The first half of the divider contains the separator "<<<<<<< HEAD" which means that the contents are from the current branch where HEAD points to - master. The second half of the divider contains the separator ">>>>>>> branch" which means the contents are from the second branch - ‘branch’. Now we need to decide whether we need to keep the first half or the second half or both the changes.

5. For the first conflict, we are deciding to keep the branch code and for the second conflict, we will keep the HEAD code. So we modify the file accordingly by keeping only the desired contents and removing the divider "=======" notation, and the separators "<<<<<<< HEAD" and ">>>>>>> branch".

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/calculator-vuejs/public
(master|MERGING)
$ vi index.html

Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/calculator-vuejs/public (ma
ster|MERGING)
$ cat index.html
<!DOCTYPE html>
<html lang="">
  <head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>

      <strong>We are very sorry but <%= htmlWebpackPlugin.options.title %> d
oesn't work properly without JavaScript enabled. Please enable it to continu
e.</strong>

    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

6. Now commit the changes and display the history.

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/calculator-vuejs/public (ma
ster|MERGING)
$ git add .

Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/calculator-vuejs/public (ma
ster|MERGING)
$ git commit -m "merge commit"
[master 1ed9fda] merge commit

Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/calculator-vuejs/public (ma
ster)
$ git log --onrline --all --graph
fatal: unrecognized argument: --onrline

Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/calculator-vuejs/public (ma
ster)
$ git log --oneline --all --graph
*   1ed9fda (HEAD -> master) merge commit
|\
| * 4efde75 (origin/branch, branch) Removed metaChar and modified user respo
nses
* | 461c9ca (origin/master, origin/HEAD) changed scentence
* |   8926d10 resolved conflict
```

Practical No. 4

Aim: Reset and Revert

Reset is the command we use when we want to move the repository back to a previous **commit**, discarding any changes made after that **commit**.

1. We need to find the point we want to return to. To do that, we need to go through the **log**.

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/G08 (master)
$ git log --oneline
00fe75c (HEAD -> master, origin/master, origin/HEAD) trying reset
d5def24 (branch1) after adding img folder to gitignore
45ece43 after gitignore
25fce69 Added Class Def to hello.cpp
12d2537  Second commit|added comment
7b973af First Commit
```

2. We want to return to the **commit**: d5def24 after adding img. We **reset** our repository back to the specific commit using **git reset *commithash*** (***commithash*** being the first 7 characters of the commit hash we found in the **log**):

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/G08 (master)
$ git reset d5def24
Unstaged changes after reset:
M       h1.html
```

3. Check the **log** again:

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/G08 (master)
$ git log --oneline
d5def24 (HEAD -> master, branch1) after adding img folder to gitignore
45ece43 after gitignore
25fce69 Added Class Def to hello.cpp
12d2537  Second commit|added comment
7b973af First Commit
```

Revert is the command we use when we want to take a previous **commit** and add it as a new **commit**, keeping the log intact.

1. Make a new **commit**, where we have "accidentally" deleted a file

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/G08 (master)
$ git commit -m "example of revert"
[master bc050ba] example of revert
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 sample.js
```

2. Now we have a part in our **commit** history we want to go back to. Let's try and do that with **revert**. We want to revert to the previous commit: bc050ba

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/G08 (master)
$ git log --oneline
bc050ba (HEAD -> master) example of revert
d5def24 (branch1) after adding img folder to gitignore
45ece43 after gitignore
25fce69 Added Class Def to hello.cpp
12d2537  Second commit|added comment
7b973af First Commit
```

3. We revert the latest commit using git revert HEAD (revert the latest change, and then commit), adding the option --no-edit to skip the commit message editor (getting the default revert message):

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/G08 (master)
$ git revert HEAD --no-edit
[master c7c404f] Revert "example of revert"
 Date: Sat Jun 4 00:48:46 2022 +0530
 2 files changed, 1 insertion(+), 2 deletions(-)
 delete mode 100644 sample.js
```

4. Now check the **log** again

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/G08 (master)
$ git log --oneline
c7c404f (HEAD -> master) Revert "example of revert"
bc050ba example of revert
d5def24 (branch1) after adding img folder to gitignore
45ece43 after gitignore
25fce69 Added Class Def to hello.cpp
12d2537  Second commit|added comment
7b973af First Commit
```

Task 2

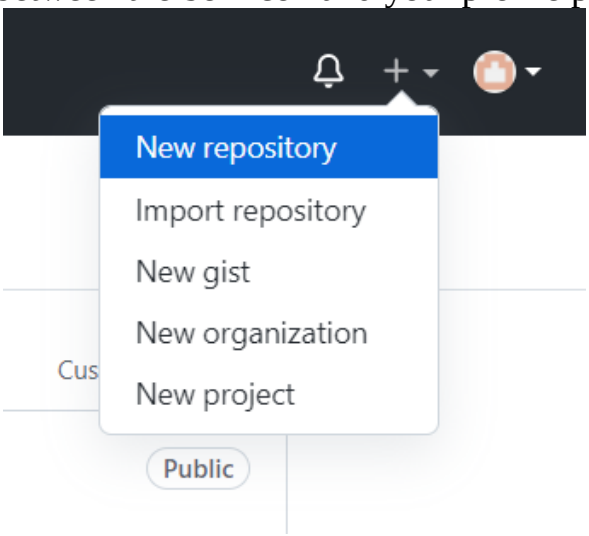
Practical No. 01

Aim: Create a distributed repository and add members in project team

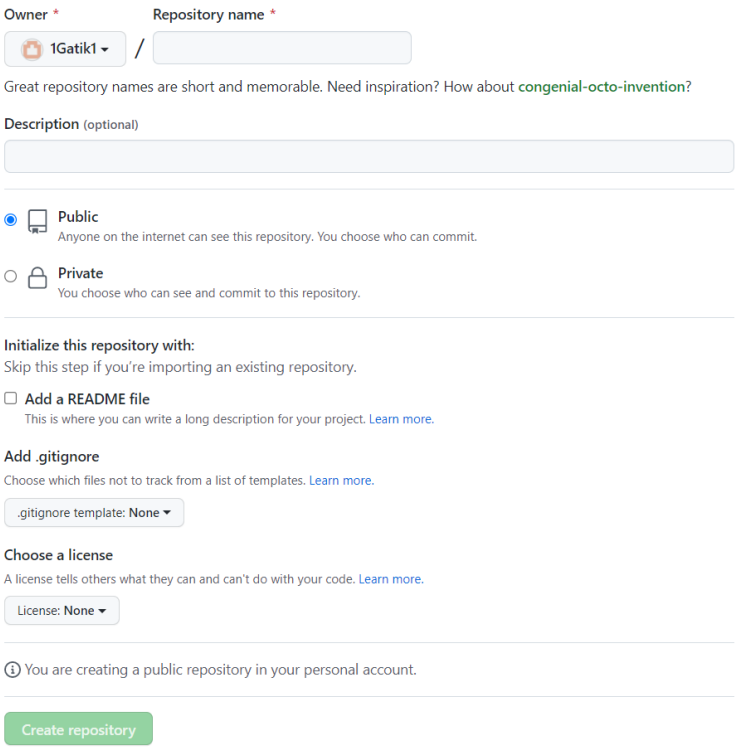
Distributed Repository:
In centralized approach, there is a single source or a central copy of your repository. You have to grant permission to other team members/ add them as collaborators to this central repository, for them to start contributing to your central repository. Whereas in distributed approach everyone clones 'a copy of the source repository' also known as 'forking'. In this approach, all the collaborators will 'fork' the source repo. Upon forking every member gets their own copy of the source repository in their GitHub account. Collaborators can then go ahead and clone their 'forked repository' on their local machines.

Creating a new Repository:

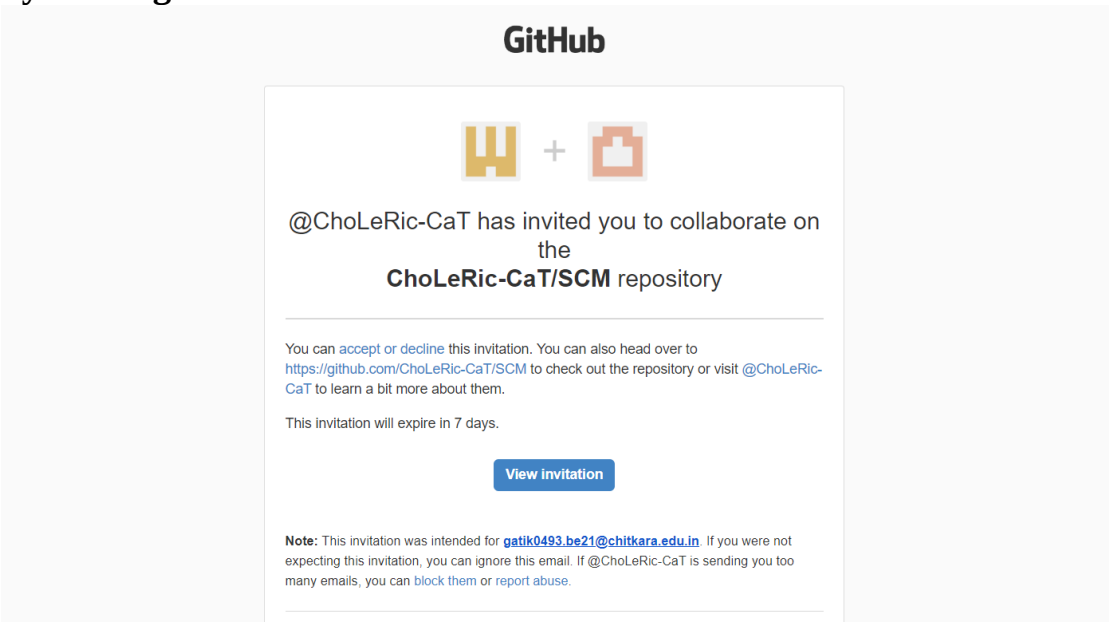
1. In the top right corner of GitHub.com, click on the plus symbol between the bell icon and your profile photo, then click new repository.



2. Enter the Repository name and add description if you want. Select if you want the repository to be public or private. Finally, click on "Create Repository".



We can now add our teammates as collaborators for our repository by sending them invites like the one below:



TEAM MEMBERS :

- Abhinn Singh Bisht 2110990056
- Dhruv Jain 2110990443
- Ankusha 2110990209
- Aayush 2110990030

Practical No. 02

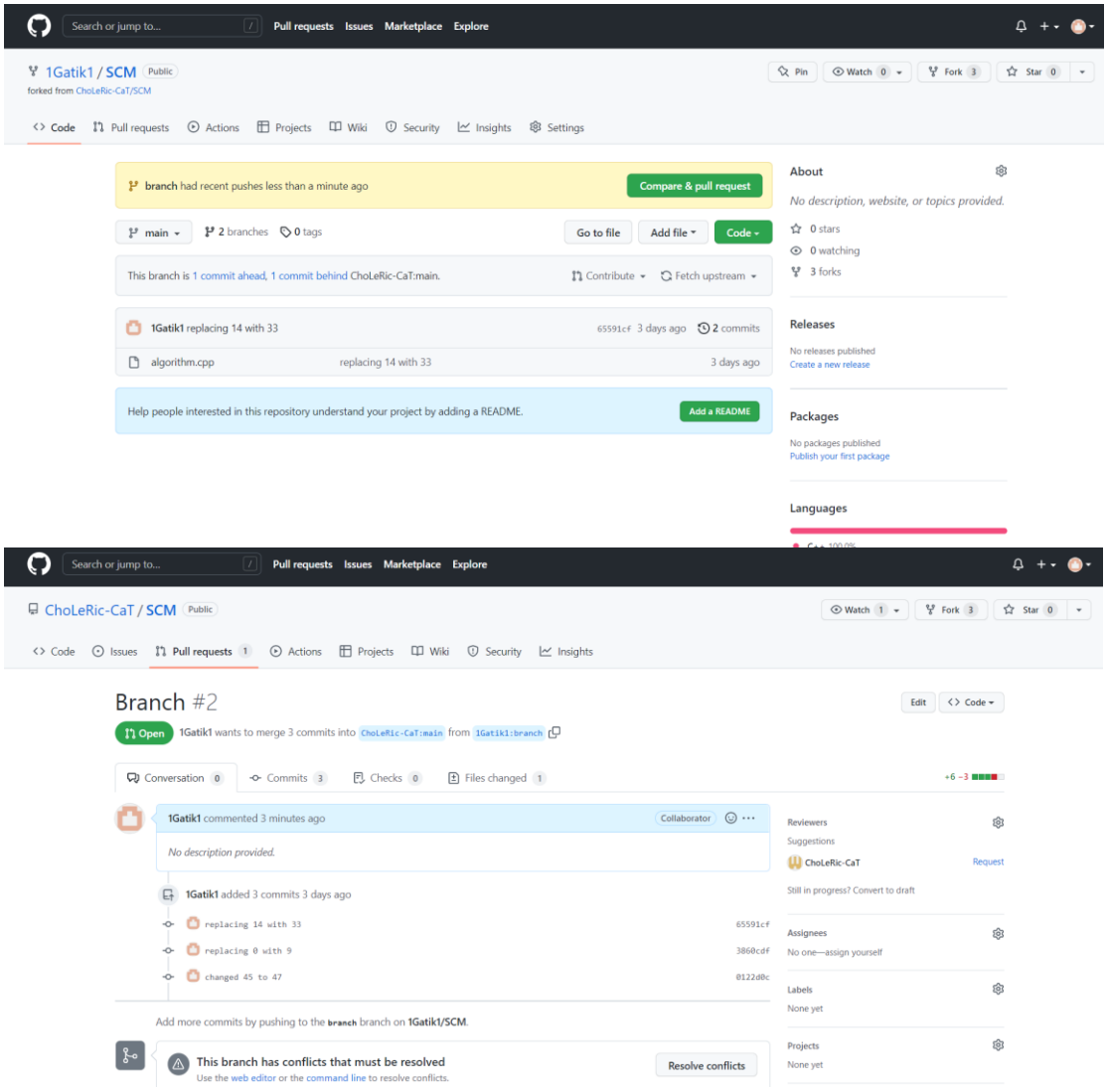
Aim: Opening and closing a pull request

Pull Requests are the heart of GitHub collaboration. With a pull request you are proposing that your changes should be merged (pulled in) with the master.
Pull requests show content differences, changes, additions, and subtractions in colors (green and red).

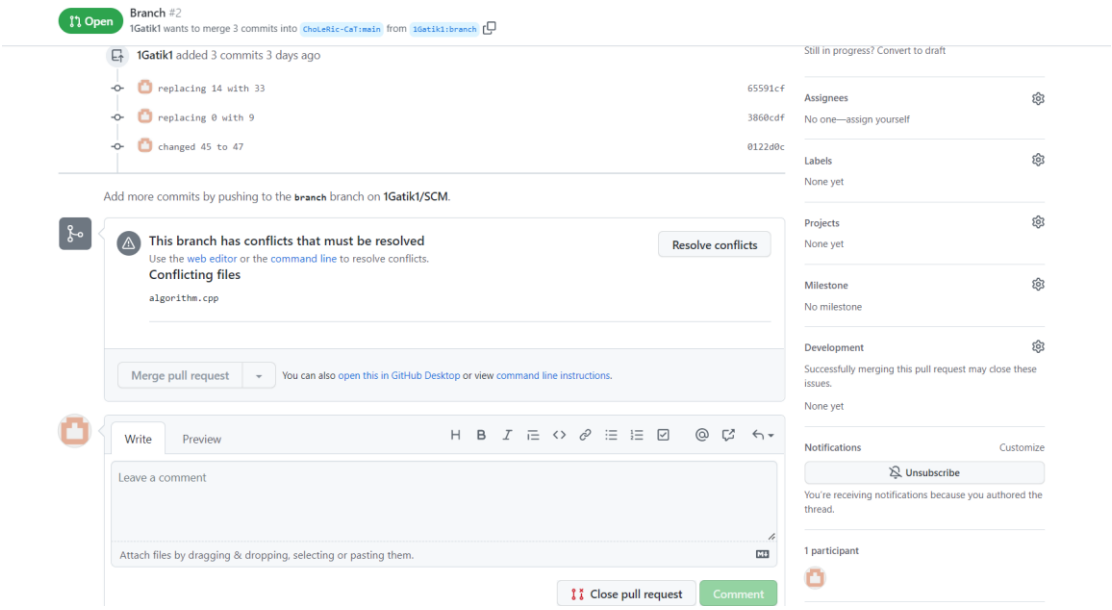
1. To open a pull request you have to make a commit in a branch of your teammate's repository.

```
Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/SCM (branch)
$ vi algorithm.cpp

Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/SCM (branch)
$ git status
On branch branch
Your branch is up to date with 'origin/branch'.

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:   algorithm.cpp

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

Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/SCM (branch)
$ git add .

Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/SCM (branch)
$ git commit -m "changed 45 to 47"
[branch 0122d0c] changed 45 to 47
 1 file changed, 3 insertions(+), 2 deletions(-)

Alok Pasi@LAPTOP-ANRC6LJM MINGW64 /d/SCM/Project/SCM (branch)
$ git push -u origin branch
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 313 bytes | 313.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/1Gatik1/SCM
   3860cdf..0122d0c  branch -> branch
branch 'branch' set up to track 'origin/branch'.
```

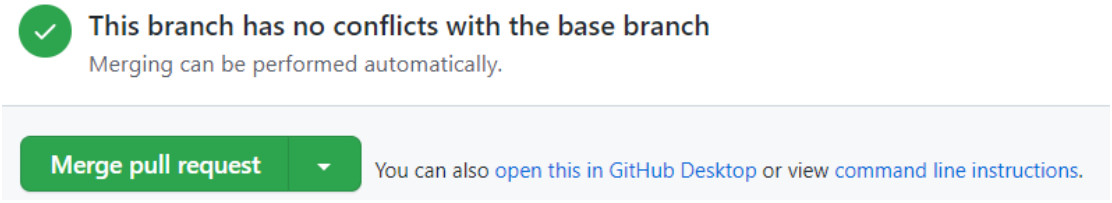
2. After pushing our commit we can now open a pull request.



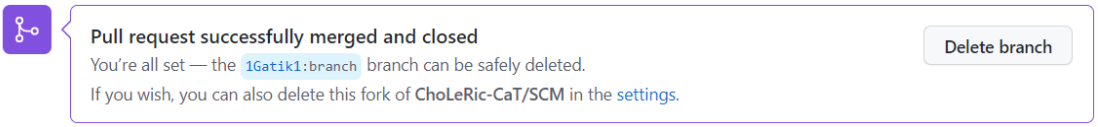
3. We cannot merge these branches due to some conflicts we have to first resolve them.



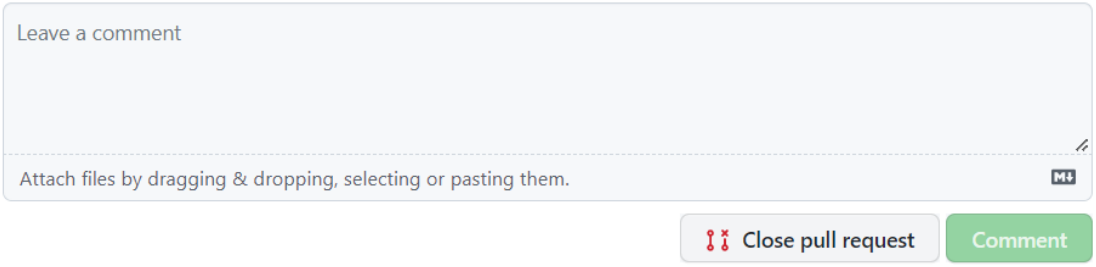
4. We can now merge branches.



5. After merging you can delete the branch by clicking a “Delete branch” button.



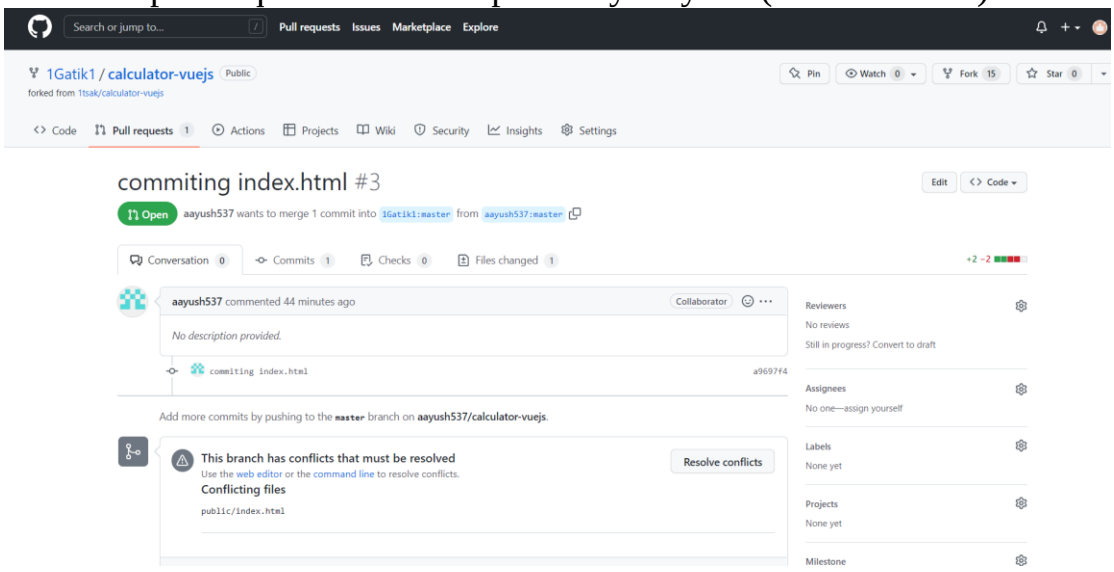
6. You can also choose “close pull request” the pull request will be closed without attempting to merge the source branch into the destination branch. This option does not provide a way to delete the source branch as part of closing the pull request, but you can do it yourself after the request is closed.



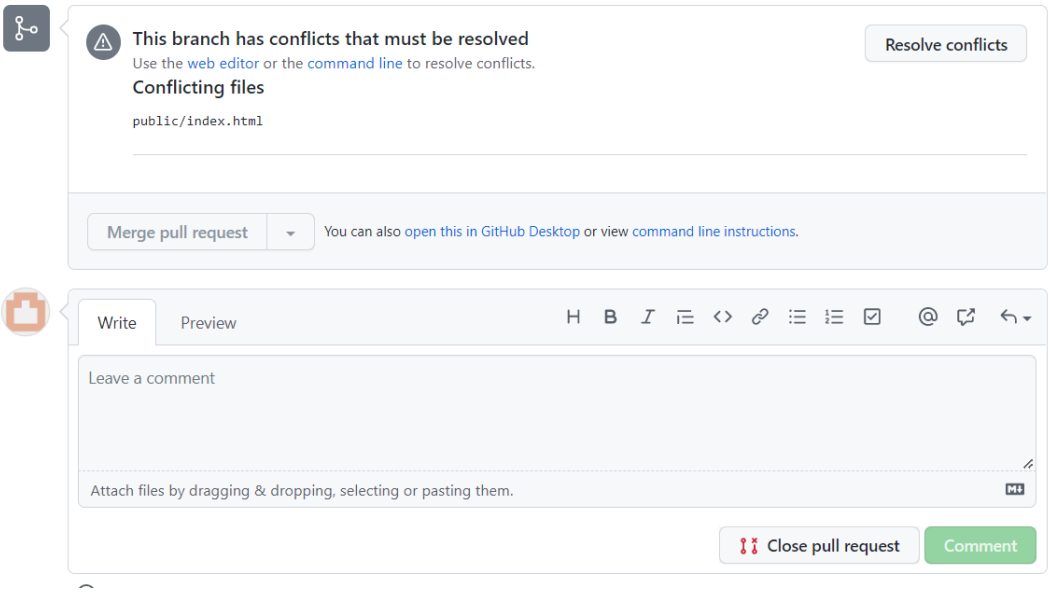
Practical No. 03

Aim: Each project member shall create a pull request on a team members Repo and close pull requests generated by team members on own Repo as a maintainer.

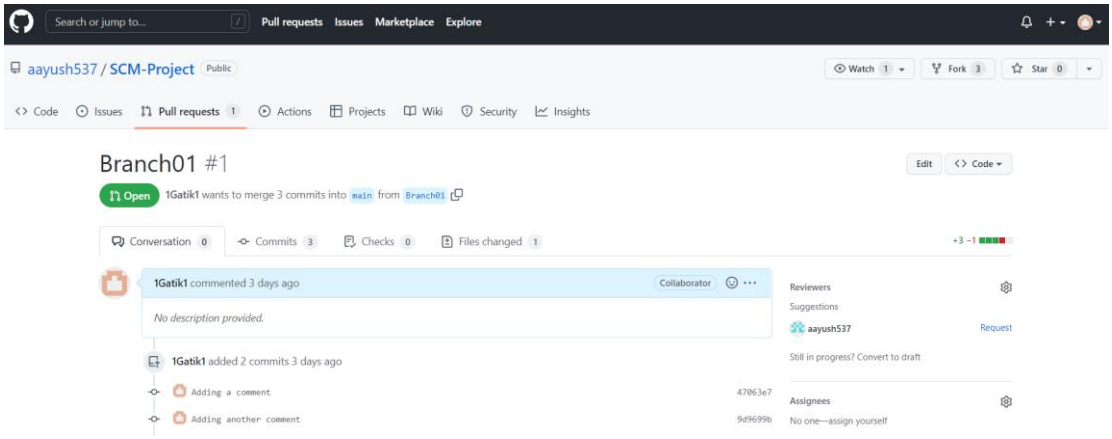
1. Here a pull request has been opened by Aayush (team member).



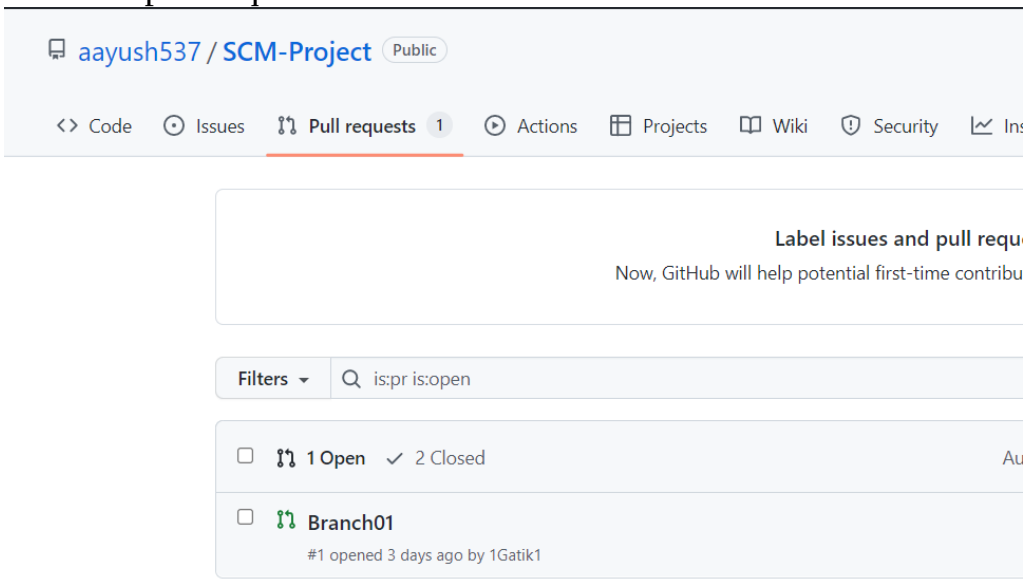
2. Now I, as the maintainer of the repo, can merge or close the pull request.



3. Here I have opened a pull request on my team member's repo.



4. Now the maintainer (Teammate Aayush) will decide to either merge or close the pull request.



Practical No. 04

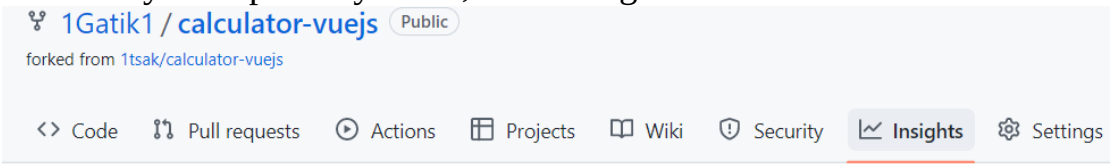
Aim: Publish and print network graphs

The network graph is one of the useful features 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.

Accessing the network graph

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

2. Under your repository name, click Insights.



3. In the left sidebar, click Network